EFFIZIENTE IMPLEMENTIERUNG EINGEBETTETER RUNGE-KUTTA-VERFAHREN DURCH AUSNUTZUNG DER SPEICHERZUGRIFFSLOKALITÄT

Von der Universität Bayreuth
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

von

Matthias Korch

aus Weißenfels

1. Gutachter:	Prof. Dr. Thomas Rauber Universität Bayreuth, Deutschland
2. Gutachter:	Prof. Dr. Christoph Kessler Universität Linköping, Schweden

Tag der Einreichung:	18. Juli 2006
Tag des Kolloqiums:	1. Dezember 2006

Zusammenfassung

Eingebettete Runge-Kutta-Verfahren zählen zu den numerischen Lösungsverfahren für nichtsteife Anfangswertprobleme gewöhnlicher Differentialgleichungssysteme. Sie werden in der Praxis häufig eingesetzt, da sie gute numerische Eigenschaften aufweisen, eine effiziente Steuerung der Schrittweite ermöglichen und aufgrund ihrer Berechnungsstruktur oft schneller die gesuchte Lösung berechnen können als alternative Verfahren. Der erforderliche Berechnungsaufwand ist dennoch sehr hoch, insbesondere für Systeme großer Dimension. Diese Arbeit beschäftigt sich mit der effizienten Implementierung eingebetteter Runge-Kutta-Verfahren. Das Ziel ist es, durch eine möglichst gute Ausnutzung der Leistungsfähigkeit moderner sequentieller und paralleler Rechnersysteme die erforderliche Berechnungszeit weitestgehend zu reduzieren. Der wichtigste Ansatzpunkt dazu ist die Optimierung der Speicherzugriffslokalität, da die Programmlaufzeit auf modernen Rechnersystemen häufig durch Wartezeiten aufgrund ausstehender Speichertransaktionen bestimmt wird.

Diesbezüglich werden zunächst verschiedene allgemeine Implementierungsvarianten für sequentielle Rechnersysteme beschrieben und ausführlich hinsichtlich ihres Lokalitätsverhaltens untersucht. Die Untersuchungen zeigen, daß die Schleifenstruktur der Implementierungen aufgrund des daraus hervorgehenden Speicherzugriffsmusters einen signifikanten Einfluß auf die Anzahl der erzeugten Cache-Fehlzugriffe und somit auch auf die Laufzeit ausübt. In Anlehnung an die sequentiellen Implementierungsvarianten werden im Anschluß allgemeine parallele Implementierungsvarianten für verteilten und für gemeinsamen Adreßraum, die das Parallelitätspotential bezüglich der Komponenten des Differentialgleichungssystems ausnutzen, entwickelt und ebenfalls hinsichtlich ihres Lokalitätsverhaltens und ihrer Skalierbarkeit untersucht. Die durchgeführten Untersuchungen machen deutlich, daß das Lokalitätsverhalten einen großen Einfluß auf die Skalierbarkeit der Implementierungen für gemeinsamen Adreßraum besitzt, während die Skalierbarkeit der Implementierungen für verteilten Adreßraum in der Regel durch die Kosten der Kommunikationsoperationen begrenzt wird. Als eine Möglichkeit zur Verbesserung der Skalierbarkeit wird der Einsatz von Techniken zur dynamischen Lastbalancierung untersucht. Der Einsatz solcher Techniken ist immer dann sinnvoll, wenn eine ungleichförmige Verteilung der Funktionsauswertungskosten bezüglich der einzelnen Komponenten des Differentialgleichungssystems vorliegt oder die Berechnung auf einem heterogenen Rechnersystem durchgeführt wird. Die durchgeführten Laufzeitexperimente zeigen, daß unter bestimmten Voraussetzungen aber auch für Differentialgleichungssysteme mit nahezu ausgeglichenen Funktionsauswertungskosten ein Laufzeitgewinn erzielt werden kann.

Da allgemeine Implementierungen weder ein optimales Lokalitätsverhalten besitzen noch eine zufriedenstellende Skalierbarkeit erreichen können, werden Möglichkeiten untersucht, Lokalität und Skalierbarkeit durch eine Spezialisierung bezüglich des Differentialgleichungssystems zu verbessern. Es wird ein blockbasierter Pipelining-Algorithmus vorgestellt, der unter der Voraussetzung, daß die rechte Seite eine beschränkte Zugriffsdistanz besitzt, bei der Integration von Differentialgleichungssystemen mit großer Dimension ein besseres Lokalitätsverhalten als die allgemeinen Implementierungen aufweist. Darüber hinaus kann der Pipelining-Algorithmus mit einem geringen Speicherplatzbedarf realisiert werden. Er eignet sich deshalb gut für die Integration mittels Linienmethode semi-diskretisierter partieller Differentialgleichungssysteme, die eine beschränkte Zugriffsdistanz besitzen und häufig einen hohen Speicherplatzbedarf aufweisen. Weiterhin ermöglicht der Pipelining-Algorithmus das frühzeitige Erkennen von Zeitschritten, die verworfen werden müssen. Bei einer parallelen Implementierung für verteilten Adreßraum kann die beschränkte Zugriffsdistanz ausgenutzt werden, um den Kommunikationsaufwand zu reduzieren, indem Einzeltransferoperationen anstelle von globalen Kommunikationsoperationen eingesetzt werden und Datenübertragungszeiten durch Berechnungen überdeckt werden. Von dem verbesserten Lokalitätsverhalten des Pipelining-Algorithmus können sowohl parallele Implementierungen für gemeinsamen Adreßraum als auch parallele Implementierungen für verteilten Adreßraum profitieren. Der zur Realisierung des Pipelining-Algorithmus verfolgte Ansatz läßt sich auf weitere Verfahren, wie z. B. iterierte Runge-Kutta-Verfahren, übertragen.

Efficient implementation of embedded Runge-Kutta methods through exploitation of the locality of memory references

Abstract

Embedded Runge-Kutta methods are among the most popular numerical solution methods for non-stiff initial value problems of ordinary differential equations. While possessing a simple computational structure, they provide desirable numerical properties and can adapt the step size efficiently. Therefore, embedded Runge-Kutta methods can often compute the solution function faster than other solution methods. However, the computation time required can be very large, particularly for systems with a high dimension. This work considers the efficient implementation of embedded Runge-Kutta methods. It aims at a reduction of the execution time by exploiting the capabilities of modern sequential and parallel computer systems. The most important approach followed is the optimization of the locality of memory references, because on modern computer systems the execution time is often determined by waiting times due to outstanding memory transactions.

In this context, several general implementation variants for sequential computer systems are described and investigated with particular regard to the locality behavior. The investigations show that the loop structure of the implementations, which is responsible for the memory reference pattern, has a significant influence on the number of generated cache misses and, hence, on the execution time. Then, using the sequential implementations as a starting point, general parallel implementations for distributed and shared address space are developed, which exploit the potential for parallelism across the system, and their locality and scalability behavior is investigated. The investigations indicate that the locality behavior has a large influence on the scalability of the implementations for shared address space, while the scalability of the implementations for distributed address space is usually limited by the costs of the communication operations. As one possibility to improve the scalability, the applicability of dynamic load balancing techniques is investigated. Such techniques are suitable if the function evaluation costs of the individual components of the ordinary differential equation system are characterized by an unequal distribution or if the integration is performed on a heterogeneous computer system. The runtime experiments performed show that under certain conditions the run time can be reduced even for systems with a nearly equal distribution of the function evaluation costs.

Since the general implementations neither possess an optimal locality behavior nor obtain a satisfying scalability, methods are investigated that improve locality and scalability by exploiting special properties of the ordinary differential equation system. A block-based pipelining algorithm is presented, which—under the precondition of a limited access distance of the right hand side function—shows a better locality behavior and a higher scalability when systems of high dimension are integrated. Moreover, the pipelining algorithm can be realized with low storage requirements. Therefore, it is particularly suitable for the integration of semi-discretized partial differential equations that have been derived by the method of lines, which possess a limited access distance and often demand for high storage space. Further, the pipelining algorithm enables the early detection of time steps that will have to be rejected. A parallel implementation for distributed address space can exploit a limited access distance to reduce the communication costs by using single transfer operations instead of global communication operations and by overlapping data transfer times by computations. The better locality behavior of the pipelining algorithm can speed up implementations for shared address space as well as implementations for distributed address space. The approach followed to realize the pipelining algorithm can also be applied to other solution methods, e.g., iterated Runge-Kutta methods.

Zugehörige Veröffentlichungen

In diese Arbeit flossen Erkenntnisse aus folgenden Veröffentlichungen des Autors ein:

Hoffmann u. a. 2004a
HOFFMANN, Ralf; KORCH, Matthias; RAUBER, Thomas: Performance Evaluation of Task Pools Based on Hardware Synchronization. In: *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2004, S. 44

Hoffmann u. a. 2004b
HOFFMANN, Ralf; KORCH, Matthias; RAUBER, Thomas: Using Hardware Operations to Reduce the Synchronization Overhead of Task Pools. In: *Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)*. Montreal, Quebec, Canada, August 2004, S. 241–249

Korch u. Rauber 2002
KORCH, Matthias; RAUBER, Thomas: Evaluation of Task Pools for the Implementation of Parallel Irregular Algorithms. In: *Proceedings of the 2002 ICPP Workshops*, IEEE Computer Society Press, August 2002, S. 597–604

Korch u. Rauber 2003
KORCH, Matthias; RAUBER, Thomas: Scalable Parallel RK Solvers for ODEs Derived by the Method of Lines. In: *Euro-Par 2003. Parallel Processing (LNCS 2790)*, Springer-Verlag, August 2003, S. 830–839

Korch u. Rauber 2004a
KORCH, Matthias; RAUBER, Thomas: Comparison of Parallel Implementations of Runge-Kutta Solvers: Message Passing vs. Threads. In: *Parallel Computing: Software Technology, Algorithms, Architectures & Applications – Proceedings of the International Conference ParCo2003, Dresden, Germany* Bd. 13. Elsevier, September 2004, S. 209–216

Korch u. Rauber 2004b
KORCH, Matthias; RAUBER, Thomas: A comparison of task pools for dynamic load balancing of irregular algorithms. In: *Concurrency and Computation: Practice and Experience* 16 (2004), Januar, S. 1–47

Korch u. Rauber 2006a
KORCH, Matthias; RAUBER, Thomas: Applicability of Load Balancing Strategies to Data-Parallel Embedded Runge-Kutta Integrators. In: *Euro-Par 2006. Parallel Processing (LNCS 4128)*, Springer-Verlag, August 2006, S. 720–729

Korch u. Rauber 2006b
KORCH, Matthias; RAUBER, Thomas: Optimizing Locality and Scalability of Embedded Runge-Kutta Solvers Using Block-Based Pipelining. In: *Journal of Parallel and Distributed Computing* 6 (2006), März, Nr. 3, S. 444–468

Korch u. Rauber 2006c
KORCH, Matthias; RAUBER, Thomas: Simulation-based analysis of parallel Runge-Kutta solvers. In: *Applied Parallel Computing: State of the Art in Scientific Computing. 7th International Workshop, PARA 2004, Lyngby, Denmark, June 2004. Revised Selected Papers*, Springer-Verlag, 2006 (LNCS 3732), S. 1105–1114

Korch u. a. 2002
KORCH, Matthias; RAUBER, Thomas; RÜNGER, Gudula: Pipelining for Locality Improvement in RK Methods. In: *Euro-Par 2002. Parallel Processing (LNCS 2400)*, Springer-Verlag, 2002, S. 724–733

Korch u. a. 2003
KORCH, Matthias; RAUBER, Thomas; RÜNGER, Gudula: Performance Optimization of RK Methods Using Block-based Pipelining. In: GETOV, V. (Hrsg.); GERNDT, M. (Hrsg.); HOISIE, A. (Hrsg.); MALONY, A. (Hrsg.); MILLER, B. (Hrsg.): *Performance Analysis and Grid Computing*. Kluwer Academic Publishers, Oktober 2003, S. 41–56

Zugehörige Abschlußarbeiten

Im Zusammenhang mit dieser Arbeit wurden vom Autor folgende Diplom- und Bachelor-Arbeiten vorgeschlagen und betreut:

Behrend 2005
BEHREND, Stefan: *Implementierung eingebetteter Runge-Kutta-Verfahren für SMP-Cluster*, Universität Bayreuth, Bachelor-Thesis, August 2005

Gerkhardt 2005
GERKHARDT, Ekaterina: *Lokalitätsoptimierung iterierter Runge-Kutta-Verfahren für gewöhnliche Differentialgleichungssysteme mit beschränkter Zugriffsdistanz*, Universität Bayreuth, Bachelor-Thesis, November 2005

Hoffmann 2003
HOFFMANN, Ralf: *Reduzierung des Synchronisations- und Verwaltungsoverheads von Taskpools auf Shared-Memory-Systemen*, Martin-Luther-Universität Halle-Wittenberg, Diplomarbeit, April 2003

Danksagung

Bedanken möchte ich mich zunächst bei Prof. Dr. Thomas Rauber für die Betreuung dieser Dissertation und für seine Unterstützung bei der Vorbereitung und der Veröffentlichung der zugehörigen Artikel. Nicht zuletzt möchte ich mich bei ihm auch dafür bedanken, daß er mir die Möglichkeit geboten hat, nach Abschluß meines Studiums am DFG-Projekt „ODE-Skalierbarkeit“ mitzuarbeiten. Dadurch lernte ich die wissenschaftliche Arbeitswelt kennen, und es entstand mein Wunsch zu promovieren.

Bedanken möchte ich mich ebenfalls bei Prof. Dr. Gudula Rünger vom Lehrstuhl für praktische Informatik der TU Chemnitz für die gute Zusammenarbeit bei der Veröffentlichung meiner ersten Artikel, die sich speziell mit Lokalitätsoptimierungen für eingebettete Runge-Kutta-Verfahren befaßten.

Weiterhin bedanke ich mich beim John von Neumann Institut für Computing im Forschungszentrums Jülich sowie dem Universitätsrechenzentrum der Martin-Luther-Universität Halle-Wittenberg für die Bereitstellung von Rechenzeit auf ihren jeweiligen Rechnersystemen. Insbesondere bedanke ich mich bei Dr. Gerald Möbius und Hans-Jürgen Walter vom Universitätsrechenzentrum der Martin-Luther-Universität Halle-Wittenberg für ihre Unterstützung bei der Durchführung von Messungen.

Mein Dank gilt weiterhin all meinen Arbeitskollegen – sowohl den Mitarbeitern des Institus für Informatik der Martin-Luther-Universität Halle-Wittenberg als auch allen Mitarbeitern des Lehrstuhls für Angewandte Informatik 2 der Universität Bayreuth – für die angenehme Arbeitsathmosphäre, die sie geschaffen haben. Besonders bedanke ich mich bei Marco Höbbel und Ralf Hoffmann, die einen direkten Beitrag zu dieser Arbeit geleistet haben. Marco danke ich dafür, daß er den von ihm entwickelten Cache-Simulator für die Verifikation von Simulationsexperimenten, die zuvor mit einer kommerziellen Simulationssoftware durchgeführt wurden, zur Verfügung gestellt und mich bei der Durchführung der Experimente unterstützt hat. Ralf danke ich für die Bereitstellung der Meßdaten zur Veranschaulichung der Skalierbarkeit eines Spin-Locks.

Ein wichtiger Beitrag zur orthographischen und grammatikalischen Qualität dieser Arbeit wurde von Judith Müller geleistet, die große Teile der Arbeit nach derartigen Mißgeschicken durchsucht hat, wofür ich mich hiermit bedanken möchte.

Auch wenn ich sie hier zuletzt nenne, so bin ich doch vor allem meinen Eltern dankbar. Ohne ihre Hilfe und Unterstützung hätte ich meinen bisherigen Lebens- und Ausbildungsweg nicht gehen können.

Inhaltsverzeichnis

1 Einführung

Zur mathematischen Beschreibung von Vorgängen in Natur und Technik, in denen mehrere Größen miteinander in Wechselwirkung stehen, benötigt man Gleichungssysteme, die diese Wechselwirkungen beschreiben. Spielen für einen Vorgang auch Änderungsraten von Größen eine Rolle, führt dies zu einem Differentialgleichungssystem. Dies ist z. B. häufig der Fall, wenn man den zeitlichen Fortschritt eines Systems simulieren möchte. Solche Simulationen sind in der heutigen Zeit von großer Bedeutung, sowohl für die Grundlagenforschung als auch für industrielle Anwendungen. Als Beispiele sollen hier die Arbeit von Harder u. a. (2004) aus dem Bereich der Geowissenschaften sowie die Arbeiten von Kleber (2001) und Kelm u. Rettig (2004) aus dem Fahrzeugbau genannt sein.

Da die analytische Lösung von Differentialgleichungssystemen in vielen Fällen nicht möglich ist, weil entweder keine geschlossene Lösung existiert oder der Aufwand zum Finden einer solchen Lösung unverhältnismäßig hoch wäre, werden in der Praxis häufig numerische Lösungsverfahren eingesetzt, die eine Näherungslösung innerhalb einer vorgegebenen Genauigkeitsschranke berechnen. Die numerische Lösung von Differentialgleichungssystemen stellt jedoch enorme Anforderungen an die Rechenleistung der verwendeten Computersysteme. Ein Grund dafür ist, daß bereits eine geringe Erhöhung der Genauigkeitsforderung oder auch eine Veränderung oder Hinzunahme anderer Parameter den Rechenzeitbedarf vervielfachen kann. So wird beispielsweise die Genauigkeit unserer heutigen Wettervorhersage, die u. a. von der Auflösung des zur Wettersimulation benutzten Ortsgitters abhängt, wesentlich durch die verfügbare Rechenkapazität begrenzt (vgl. Majewski u. a. 2003). Besonders deutlich wird der hohe Rechenzeitbedarf, wenn man die Anwendungsgebiete heutiger Supercomputer betrachtet. So wurde z. B. der japanische „Earth Simulator“ (siehe Earth Simulator Center/JAMSTEC 2004), der bis zum Juni 2004 die Top-500-Liste der Supercomputer anführte, eigens für die Durchführung von Klimasimulationen gebaut. Und auch der z. Z. schnellste deutsche Supercomputer, der am John-von-Neumann-Institut für Computing des Forschungszentrums Jülich installiert ist, wird überwiegend zur Berechnung von Simulationen aus den Bereichen Astrophysik, Elementar- und Vielteilchenphysik, Polymere, Chemie und Umwelt genutzt (siehe NIC-Broschüre 2005).

Aufgrund der hohen Anforderungen an die Rechenleistung sind die bekannten Lösungsverfahren für Differentialgleichungssysteme einer ständigen Weiterentwicklung unterzogen, und es wird nach neuen Verfahren gesucht, damit Lösungen zukünftig schneller berechnet werden können. In der vorliegenden Dissertation soll eine spezielle Klasse von Lösungsverfahren für nichtsteife Anfangswertprobleme gewöhnlicher Differentialgleichungen, die sogenannten eingebetteten Runge-Kutta-Verfahren, betrachtet und untersucht werden. Ziel dabei ist es, durch Verbesserung der Speicherzugriffslokalität die Laufzeit zu verbessern und effiziente parallele Implementierungen zu finden, welche eine verbesserte Ausnutzung der Speicherhierarchien moderner Parallelrechner erlauben.

1.1 Numerische Lösung gewöhnlicher Differentialgleichungen

Diese Arbeit betrachtet Lösungsverfahren für *Anfangswertprobleme expliziter gewöhnlicher Differentialgleichungssysteme erster Ordnung*. In *nichtautonomer* Form haben diese folgende Gestalt:

$$\begin{aligned} \mathbf{y}'(t) &= \mathbf{f}(t, \mathbf{y}(t)) \ , \\ \mathbf{y}(t_0) &= \mathbf{y}_0 \ . \end{aligned} \tag{1.1}$$

Die Variable $t \in \mathbb{R}$ wird häufig als *Zeit* bezeichnet, da viele gewöhnliche Differentialgleichungssysteme die zeitliche Veränderung eines physikalischen Systems beschreiben. Die Lösung des Anfangswertproblems

ist die Funktion $\mathbf{y} : \mathbb{R} \to \mathbb{R}^n$, deren erste Ableitung bezüglich t durch die Funktion $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ beschrieben wird und die der *Anfangsbedingung* $\mathbf{y}(t_0) = \mathbf{y}_0$ mit gegebenem *Anfangswert* $\mathbf{y}_0 \in \mathbb{R}^n$ genügt. Die Begriffe *Differentialgleichung* und *Differentialgleichungssystem* werden oft synonym verwendet, da man von einer Gleichung sprechen kann, wenn man sich auf die Vektorschreibweise bezieht, die Bezeichnung als Gleichungssystem aber ebenfalls zutreffend ist, wenn man die einzelnen Vektorkomponenten betrachtet. Die englische Bezeichnung für *gewöhnliche Differentialgleichung*, die in der Fachliteratur häufig vorzufinden ist, lautet *ordinary differential equation*, abgekürzt: *ODE*. Die englische Abkürzung wird auch in dieser Arbeit an verschiedenen Stellen verwendet.

Mit Hilfe von Lösungsverfahren für (1.1) kann auch eine Reihe anderer Anfangswertaufgaben behandelt werden. *Autonome* Anfangswertprobleme

$$\begin{aligned} \mathbf{y}'(t) &= \mathbf{f}(\mathbf{y}(t)) \ , \\ \mathbf{y}(t_0) &= \mathbf{y}_0 \ , \end{aligned} \tag{1.2}$$

bei denen $\mathbf{f}$ nicht explizit von t abhängt, lassen sich durch die formale Einführung der Variablen t als Argument der Funktion $\mathbf{f}$ ohne weitere Modifikation lösen. Explizite Anfangswertprobleme höherer Ordnung

$$\begin{aligned} \mathbf{y}^{(m)}(t) &= \mathbf{f}(t, \mathbf{y}(t), \mathbf{y}'(t), \ldots, \mathbf{y}^{(m)}(t)) \ , \\ \mathbf{y}^{(i)}(t_0) &= \mathbf{y}_{0i} \ , \quad i = 0, \ldots, m-1 \ , \quad 1 \leq m \in \mathbb{N} \ , \end{aligned} \tag{1.3}$$

können in ein äquivalentes System der Form (1.1) der Größe $n \cdot m$ überführt werden. Algebro-Differentialgleichungen und implizite gewöhnliche Differentialgleichungen lassen sich ebenfalls in ein System der Form (1.1) überführen. Um die speziellen Eigenschaften der letztgenannten Problemklassen zu deren Lösung ausnutzen zu können, sind für diese spezialisierte Lösungsverfahren allerdings oft die geeignetere Wahl.

Die numerische Lösung von Anfangswertaufgaben der Form (1.1) auf einem vorgegebenen Intervall $[t_0, t_e]$ erfordert die Diskretisierung des Intervalls mit Hilfe eines Punktgitters

$$I = \{t_0, t_1, \ldots, t_N\} \quad \text{mit} \quad t_0 < t_1 < \cdots < t_N \leq t_e \ . \tag{1.4}$$

Berechnet wird eine Näherungslösung, d. h. eine Gitterfunktion

$$\eta(t) : I \to \mathbb{R}^n \quad \text{mit} \quad \eta(t_\kappa) \approx \mathbf{y}(t_\kappa) \quad \text{für} \quad 0 \leq \kappa \leq N, \quad \kappa \in \mathbb{N} \ , \tag{1.5}$$

welche die gesuchte Lösungsfunktion auf dem vorgegebenen Intervall approximiert. Zur Vereinfachung der Schreibweise setzen wir

$$\eta_\kappa = \eta(t_\kappa) \ .$$

Zur Berechnung der Gitterfunktion werden bei klassischen Diskretisierungsverfahren, beginnend mit $\kappa = 0$ und damit ausgehend vom bekannten Anfangswert $\eta_0 = \eta(t_0) = \mathbf{y}_0$, mehrere *Zeitschritte* $\kappa \to \kappa + 1$ ausgeführt, in denen nacheinander die Werte der Gitterfunktion an den Gitterpunkten t_κ bestimmt werden. Die Differenz

$$h_\kappa = t_{\kappa+1} - t_\kappa$$

wird dabei als *Schrittweite* bezeichnet.

Der verbleibende Teil dieses Abschnittes gibt einen Überblick über einige wichtige Verfahrensklassen, die zur Lösung gewöhnlicher Differentialgleichungssysteme eingesetzt werden.

1.1.1 Einschrittverfahren

Einschrittverfahren verwenden zur Berechnung eines neuen Approximationswertes $\eta_{\kappa+1}$ ausschließlich den Approximationswert η_κ, der im vorangegangenen Zeitschritt berechnet wurde. Sie haben folgende Gestalt:

$$\begin{aligned} \eta_{\kappa+1} &= \eta_\kappa + h_\kappa \varphi(t_\kappa, \eta_\kappa; h_\kappa; \mathbf{f}) \ , \quad \kappa = 0, \ldots, N-1 \ , \\ \eta_0 &= \mathbf{y}_0 \ . \end{aligned} \tag{1.6}$$

φ wird dabei als *Verfahrensfunktion* oder *Inkrementfunktion* bezeichnet. In Abhängigkeit davon, ob die Verfahrensfunktion zur Bestimmung von $\eta_{\kappa+1}$ auch auf $\eta_{\kappa+1}$ selbst zurückgreift oder ob sie ausschließlich η_κ verwendet, unterscheidet man zwischen *expliziten* und *impliziten* Einschrittverfahren. Da bei impliziten Verfahren der gesuchte Lösungsvektor sowohl links als auch rechts des Gleichheitszeichens auftritt, muß in der Regel in jedem Iterationsschritt ein nichtlineares Gleichungssystem gelöst werden. Explizite Verfahren können den gesuchten Lösungsvektor hingegen berechnen, indem sie lediglich eine Anzahl von Funktionsauswertungen von $\mathbf{f}$ durchführen. Allerdings haben explizite Verfahren den Nachteil, daß sie eine schlechtere numerische Stabilität besitzen, d. h., der globale Diskretisierungsfehler pflanzt sich stärker fort als bei impliziten Verfahren. Infolgedessen können bestimmte Anfangswertprobleme nicht effizient mit expliziten Verfahren gelöst werden, da bei diesen Problemen die Schrittweite h extrem klein gewählt werden müßte. Diese Probleme bezeichnet man als *steif* (Curtis u. Hirschfelder 1952).

Das einfachste Einschrittverfahren wurde von Euler (1768) beschrieben. Ausgehend von einem Wertepaar $(t_0, \mathbf{y}_0)$ wird angenommen, daß für eine kleine Änderung $\Delta t = t_1 - t_0$ die zugehörige Änderung $\Delta\mathbf{y} = \mathbf{y}_1 - \mathbf{y}_0$ durch

$$\Delta\mathbf{y} \approx \Delta t\, \mathbf{f}(t_0, \mathbf{y}_0)$$

approximiert werden kann. Daraus ergibt sich das Verfahren

$$\eta_{\kappa+1} = \eta_\kappa + h_\kappa \mathbf{f}(t_\kappa, \eta_\kappa) \tag{1.7}$$

mit der Verfahrensfunktion $\varphi(t_\kappa, \eta_\kappa; h_\kappa; \mathbf{f}) = \mathbf{f}(t_\kappa, \eta_\kappa)$, welches als *explizites Euler-Verfahren* bezeichnet wird. Verwendet man zur Approximation von $\Delta\mathbf{y}$ statt des Anstiegs im Punkt $(t_0, \mathbf{y}_0)$ den Anstieg im Punkt $(t_1, \mathbf{y}_1)$, erhält man das *implizite Euler-Verfahren*:

$$\eta_{\kappa+1} = \eta_\kappa + h_\kappa \mathbf{f}(t_{\kappa+1}, \eta_{\kappa+1}) \ . \tag{1.8}$$

Runge-Kutta-Verfahren

Die wichtigste Klasse von Einschrittverfahren geht auf die Arbeiten von Runge (1895, 1905), Heun (1900) und Kutta (1901) zurück. Die Verfahren dieser Klasse werden deshalb als *Runge-Kutta-Verfahren* (RK-Verfahren) bezeichnet. Um die Konvergenzordnung, d. h. die Genauigkeit, mit der die Näherungslösung $\eta_{\kappa+1}$ aus η_κ berechnet wird, im Vergleich zum Euler-Verfahren zu verbessern, werden in jedem Zeitschritt zusätzlich mehrere Zwischenlösungen berechnet und zur Berechnung von $\eta_{\kappa+1}$ herangezogen. Werden von einem Verfahren s Zwischenlösungen benutzt, so spricht man von einem s-stufigen Verfahren. Die Idee der Runge-Kutta-Verfahren beruht darauf, die Taylorentwicklung für $\mathbf{y}(t_\kappa + h_\kappa)$ zu approximieren, indem die in der Taylorentwicklung vorkommenden elementaren Differentiale durch eine geeignete Linearkombination von Zwischenlösungen nachgebildet werden. Ausführliche Darstellungen zur Konstruktion von Runge-Kutta-Verfahren findet man z. B. in Hairer u. a. (2000) sowie Strehmel u. Weiner (1995).

Die ersten Runge-Kutta-Verfahren waren explizite Verfahren. Eine allgemeine Formulierung, die auch implizite Verfahren umfaßt, wurde von Butcher (1963, 1964a) eingeführt:

$$\mathbf{v}_l = \mathbf{f}\left(t_\kappa + c_l h_\kappa, \eta_\kappa + h_\kappa \sum_{i=1}^{s} a_{li}\mathbf{v}_i\right) \quad \text{für} \quad l = 1, \ldots, s \ , \tag{1.9a}$$

$$\eta_{\kappa+1} = \eta_\kappa + h_\kappa \sum_{l=1}^{s} b_l \mathbf{v}_l \ . \tag{1.9b}$$

Die Koeffizienten a_{li}, b_l und c_l bestimmen das konkrete Verfahren und werden üblicherweise in Matrix- bzw. Vektorform geschrieben:

$$A = (a_{li})_{\{l,i=1,\ldots,s\}} \in \mathbb{R}^{s\times s} \ , \quad \mathbf{b} = (b_1, \ldots, b_s)^T \in \mathbb{R}^s \ , \quad \mathbf{c} = (c_1, \ldots, c_s)^T \in \mathbb{R}^s \ .$$

Die Matrix A nennt man *Verfahrensmatrix*, der Vektor $\mathbf{c}$ heißt *Knotenvektor* (auch *Stützstellenvektor*), und den Vektor $\mathbf{b}$ bezeichnet man als *Wichtungsvektor*. Die Vektoren $\mathbf{v}_1, \ldots, \mathbf{v}_s$ bezeichnen wir als *Stufenvektoren*. Bei

einem expliziten Verfahren ist A eine strikte untere Dreiecksmatrix. Andernfalls liegt ein implizites Verfahren vor. Eine gebräuchliche Darstellungsform für ein spezifisches Runge-Kutta-Verfahren ist das Butcher-Tableau:

$$\begin{array}{c|ccc} c_1 & a_{1,1} & \dots & a_{1,s} \\ \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s,1} & \dots & a_{s,s} \\ \hline & b_1 & \dots & b_s \end{array}$$

Besondere Bedeutung unter den expliziten Runge-Kutta-Verfahren besitzen die *eingebetteten Runge-Kutta-Verfahren*, weil diese eine effiziente adaptive Bestimmung der Gitterfunktion zur Diskretisierung des Integrationsintervalls erlauben. Diese spezielle Form der *Schrittweitenkontrolle* wird später in einem separaten Abschnitt besprochen.

Implizite RK-Verfahren besitzen bessere Stabilitätseigenschaften als explizite Verfahren. Sie erkaufen diesen Vorteil jedoch durch einen erheblich höheren Berechnungsaufwand, denn aufgrund der impliziten Verfahrensvorschrift ist die Lösung nichtlinearer Gleichungssysteme erforderlich. Die erste systematische Untersuchung impliziter RK-Verfahren erfolgte durch Butcher (1964a). Später wurden ausführliche theoretische Untersuchungen durchgeführt, die u. a. in den Monographien von Butcher (2003), Dekker u. Verwer (1984) und Hairer u. Wanner (2002) dargestellt werden.

Ein s-stufiges implizites RK-Verfahren kann maximal die Ordnung $2s$ besitzen. Verfahren, die diese maximale Ordnung erreichen, basieren auf der Gauß-Legendre-Quadratur und werden deshalb als Gauß-Verfahren bezeichnet (Kuntzmann 1961; Ceschino u. Kuntzmann 1963; Butcher 1964a). Die Stabilität der Gauß-Verfahren ist allerdings nicht optimal, und es gibt Verfahren mit schlechterem Verhältnis von Ordnung und Stufenzahl, die eine bessere Stabilität erreichen. Bekannte Verfahren der Ordnung $2s-1$ sind die Radau-Verfahren I und II (Butcher 1964b) sowie IA und IIA (Ehle 1968). Unter den Verfahren der Ordnung $2s-2$ sind vor allem die Lobatto-Verfahren IIIA, IIIB (Ehle 1968) und IIIC (Chipman 1971) bekannt.

Der enorme Berechnungsaufwand zur Lösung der nichtlinearen Gleichungssysteme läßt sich verringern, wenn man eine spezielle Struktur der Verfahrensmatrix A voraussetzt. Das heißt insbesondere, daß die Matrix A nicht voll besetzt ist. Derartige Verfahren können allerdings keine so hohe Ordnung im Verhältnis zur Stufenanzahl erreichen wie Verfahren mit einer vollbesetzten Verfahrensmatrix. Abhängig von den konkreten Eigenschaften der Verfahrensmatrix A unterscheidet man zwischen diagonal-impliziten RK-Verfahren (DIRK-Verfahren), bei denen A eine untere Dreiecksmatrix ist, einfach-diagonal-impliziten RK-Verfahren (SDIRK-Verfahren von englisch: *singly diagonally implicit RK methods*), bei denen zusätzlich alle Diagonalelemente gleich sind, und einfach-implizite RK-Verfahren (SIRK-Verfahren), deren Verfahrensmatrix einen s-fachen realen Eigenwert besitzt. Weiterhin existieren linear-implizite RK-Verfahren (LIRK-Verfahren), die durch die Einbeziehung der für die Lösung der nichtlinearen Gleichungssysteme erforderlichen Jacobi-Matrix in die Verfahrensvorschrift charakterisiert sind. Man unterscheidet hier zwischen W- (Steihaug u. Wolfbrandt 1979) und ROW-Methoden (Rosenbrock 1963; Wanner 1977) sowie adaptiven Runge-Kutta-Verfahren (Strehmel u. Weiner 1982).

Extrapolationsverfahren

Ausgehend von der Approximation η_κ der Lösungsfunktion zum Zeitpunkt t_κ ermitteln Extrapolationsverfahren mehrere verschiedene Approximationen für den Zeitpunkt $t_{\kappa+1}$, aus denen anschließend eine Näherungslösung mit höherer Konvergenzordnung als die einzelnen Approximationen extrapoliert wird. Dazu werden mehrere Integrationen über das Intervall $[t_\kappa, t_{\kappa+1}]$ mit unterschiedlichen Schrittweiten durchgeführt. Als Basisverfahren werden oft Verfahren der Ordnung 1 oder 2 verwendet. Abhängig von der Art des Basisverfahrens unterscheidet man explizite und implizite Extrapolationsverfahren. Für explizite Extrapolationsverfahren kommen z. B. das explizite Euler-Verfahren oder die explizite Mittelpunktregel (GBS-Verfahren, Gragg 1965; Bulirsch u. Stoer 1966) als Basisverfahren in Betracht. Als implizite Basisverfahren eignen sich u. a. die linear-implizite Mittelpunktsregel (Bader u. Deuflhard 1983) oder das linear-implizite Euler-Verfahren (Deuflhard u. Nowak 1987).

Die Schrittweiten

$$h^{(1)} < h^{(2)} < h^{(3)} < \dots$$

für die Bestimmung der einzelnen Approximationen werden durch eine aufsteigende Folge positiver ganzer Zahlen

$$n_1 < n_2 < n_3 < \cdots \tag{1.10}$$

festgelegt, indem man für $i \geq 1$

$$h^{(i)} = \frac{h_\kappa}{n_i}$$

setzt. Verwendete Folgen für (1.10) sind z. B. die *Romberg-Folge* (Romberg 1955)

$$1, 2, 4, 8, 16, 32, 64, 128, 256, 512, \ldots$$

und die *Bulirsch-Folge* (Romberg 1955)

$$1, 2, 3, 4, 6, 8, 12, 16, 24, 32, \ldots \quad .$$

Am vorteilhaftesten für ein Anfangswertproblem ist jedoch die *harmonische Folge* (vgl. Deuflhard 1983):

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots \quad .$$

Eine mit der Schrittweite $h^{(i)}$ ermittelte Näherungslösung für $\eta_{\kappa+1}$ wird mit $T_{i,1}$ bezeichnet. Zur Durchführung der Extrapolation wird ein Interpolationspolynom

$$T(h) = e_0 - e_p h^p - e_{p+1} h^{p+1} - \cdots - e_{p+k-2} h^{p+k-2}$$

ermittelt, so daß

$$T\left(h^{(i)}\right) = T_{i,1} \quad \text{für} \quad i = j, j-1, \ldots, j-k+1$$

gilt. Das numerische Ergebnis der Extrapolation ist der Wert des Interpolationspolynoms für $h = 0$:

$$T(0) = e_0 =: T_{j,k} \quad .$$

Besitzt das Basisverfahren die Ordnung 1, muß lediglich der Wert des Interpolationspolynoms

$$T(h) = e_0 - e_1 h - \cdots - e_{k-1} h^{k-1}$$

an der Stelle $h = 0$ bestimmt werden. Dazu bietet sich der Aitken-Neville-Algorithmus (Aitken 1932; Neville 1934)

$$T_{j,k+1} = T_{j,k} + \frac{T_{j,k} - T_{j-1,k}}{(n_j / n_{j-k})^2 - 1} \tag{1.11}$$

an.

Schrittweitenkontrolle

Um eine vorgegebene Genauigkeit der näherungsweise ermittelten Lösungsfunktion $\eta(t)$ zu gewährleisten, muß in jedem Zeitschritt κ die Schrittweite h_κ hinreichend klein gewählt werden. Die Schrittweite sollte allerdings auch nicht zu klein sein, denn dies würde zu einem unnötig hohen Rechenaufwand führen. Die optimale Wahl der Schrittweite hängt von den Eigenschaften des zu lösenden Anfangswertproblems ab. Da sich das Verhalten eines gewöhnlichen Differentialgleichungssystems innerhalb des betrachteten Integrationsintervalls ändern kann, führt die Wahl einer konstanten Schrittweite in der Regel nicht zur gewünschten Minimierung des Rechenaufwandes. Effiziente Implementierungen von Lösungsverfahren für gewöhnliche Differentialgleichungssysteme passen deshalb die Schrittweite nach jedem Zeitschritt derart an, daß jeweils eine möglichst große Schrittweite gewählt wird, die aber noch die Genauigkeitsforderung erfüllt.

Um eine geeignete Schrittweite festlegen zu können, ist es zunächst erforderlich, den lokalen Fehler der aktuellen Näherungslösung abzuschätzen. Dazu gibt es verschiedene Ansätze:

Richardson-Extrapolation für Runge-Kutta-Verfahren. Die Richardson-Extrapolation geht auf Arbeiten von Richardson (1910, 1927) zurück. Sie läßt sich sowohl auf explizite wie auch auf implizite Verfahren anwenden. Zunächst wird ausgehend von η_κ durch einen einzelnen Runge-Kutta-Schritt mit der Schrittweite h_κ eine Näherungslösung $\eta_{\kappa+1}^{(h_\kappa)}$ berechnet. Danach wird durch Durchführung zweier Runge-Kutta-Schritte mit der Schrittweite $\frac{h_\kappa}{2}$ eine zweite Näherungslösung $\eta_{\kappa+1}^{(2\times h_\kappa/2)}$ ermittelt. Zur Fehlerschätzung kann man z. B. die Größe

$$\epsilon = \max_{i=1,\ldots,n} \frac{\left|\eta_{\kappa+1,i}^{(h_\kappa)} - \eta_{\kappa+1,i}^{(2\times h_\kappa/2)}\right|}{(2^p-1)S_i} \tag{1.12}$$

mit den Skalierungsfaktoren

$$S_i = \mathrm{ATOL} + \mathrm{RTOL} \cdot \max\left(\left|\eta_{\kappa,i}\right|, \left|\eta_{\kappa+1,i}^{(2\times h_\kappa/2)}\right|\right) \tag{1.13}$$

definieren. ATOL und RTOL sind dabei vom Anwender vorgegebene Toleranzen, und p bezeichnet die Ordnung des Verfahrens. Die neue Schrittweite h_{neu} kann man anschließend wie folgt festlegen:

$$h_{\mathrm{neu}} = h_\kappa \cdot \min\left(\alpha_{\max}, \max\left(\alpha_{\min}, \alpha(1/\epsilon)^{1/(p+1)}\right)\right) \quad . \tag{1.14}$$

Für den Sicherheitsfaktor α wird häufig $\alpha = 0.8$, $\alpha = 0.9$ oder $\alpha = (0,25)^{1/(p+1)}$ benutzt. Gebräuchliche Werte für $\alpha_{\max}$ liegen zwischen 1,5 und 5, für $\alpha_{\min}$ zwischen 0,2 und 0,5.

Ist $\epsilon < 1$, wird der Integrationsschritt $t_\kappa \to t_{\kappa+1}$ akzeptiert, und man geht mit der extrapolierten Näherungslösung

$$\eta_{\kappa+1} = \eta_{\kappa+1}^{(2\times h_\kappa/2)} + \frac{\eta_{\kappa+1}^{(2\times h_\kappa/2)} - \eta_{\kappa+1}^{(h_\kappa)}}{2^p - 1} \tag{1.15}$$

und der Schrittweite $h_{\kappa+1} = h_{\mathrm{neu}}$ zum nächsten Integrationsschritt über. Andernfalls muß der Integrationsschritt mit Schrittweite $h_\kappa = h_{\mathrm{neu}}$ wiederholt werden.

Eingebettete Runge-Kutta-Verfahren. Bei Anwendung der Richardson-Extrapolation sind für die Berechnung von $\eta_{\kappa+1}^{(h_\kappa)}$ und $\eta_{\kappa+1}^{(2\times h_\kappa/2)}$ insgesamt $(3s-1)$ Funktionsauswertungen erforderlich. Aufgrund dieses erhöhten Berechnungsaufwandes wurden explizite Runge-Kutta-Verfahren konstruiert, die dem Schema

$$\begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{21} & & & & \\ c_3 & a_{31} & a_{32} & & & \\ \vdots & \vdots & \vdots & \ddots & & \\ c_s & a_{s,1} & a_{s,2} & \ldots & a_{s,s-1} & \\ \hline & b_1 & b_2 & \ldots & b_{s-1} & b_s \\ \hline & \hat{b}_1 & \hat{b}_2 & \ldots & \hat{b}_{s-1} & \hat{b}_s \end{array} \tag{1.16}$$

entsprechen und bereits selbst eine zweite Approximation liefern, ohne daß zusätzliche Funktionsauswertungen erforderlich sind. Die zweite Approximation

$$\hat{\eta}_{\kappa+1} = \eta_\kappa + h_\kappa \sum_{l=1}^{s} \hat{b}_l \mathbf{v}_l \tag{1.17}$$

hat dabei die Ordnung q, die sich in der Regel um 1 von der Ordnung p der Näherungslösung $\eta_{\kappa+1}$ unterscheidet, d. h. $q = p-1$ oder $q = p+1$. Die Differenz $\hat{\eta}_{\kappa+1} - \eta_{\kappa+1}$ liefert einen Näherungswert für den Hauptteil des lokalen Diskretisierungsfehlers des Runge-Kutta-Verfahrens mit der geringeren Ordnung $q^* = \min(p,q)$:

$$\epsilon = \max_{i=1,\ldots,n} \frac{|\eta_{\kappa,i} - \hat{\eta}_{\kappa+1,i}|}{S_i} \tag{1.18}$$

mit

$$S_i = \text{ATOL} + \text{RTOL} \cdot \max(|\eta_{\kappa,i}|, |\eta_{\kappa+1,i}|) \quad . \tag{1.19}$$

Die neue Schrittweite kann analog der Richardson-Extrapolation mittels (1.14) bestimmt werden, wobei p durch q^* zu ersetzen ist.

Die ersten Verfahren dieser Art wurden von Merson (1957), Ceschino (1962) und Zonneveld (1963, 1964) vorgeschlagen. Weitere Verfahren bis zu einer Ordnung von 8 wurden u. a. von Sarafyan (1966), England (1969), Fehlberg (1964, 1968, 1969), Verner (1978) und Prince u. Dormand (1981) entwickelt.

Für implizite Verfahren, insbesondere für Verfahren mit hoher Ordnung, ist die Gewinnung eines zweiten Näherungswertes der Ordnung $q \geq s$ nur durch Gewichte $\hat{b}_i$ nicht möglich. Hairer u. a. (2000) konstruieren deshalb für das Radau-IIA(3)-Verfahren einen neuen Näherungswert, der allerdings nur die Ordnung 3 besitzt. Für SIRK-Verfahren der Ordnung $p = s$ läßt sich ein eingebettetes Verfahren der Ordnung $q = p + 1$ oder $q = p - 1$ dagegen recht einfach bestimmen. Dies wurde erstmal von Burrage (1978) demonstriert. Basierend auf der dort vorgestellten Konstruktionsmethode wurde das Programmpaket STRIDE von Burrage u. a. (1980) entwickelt.

Schrittweiten- und Ordnungssteuerung bei Extrapolationsverfahren. Die Wahl der Schrittweite kann bei Extrapolationsverfahren in der gleichen Weise geschehen, wie bei eingebetteten Runge-Kutta-Verfahren. Nachdem die ersten k Zeilen des Extrapolationstableaus berechnet wurden, ist $T_{k,k}$ die Approximation mit der höchsten Ordnung, nämlich $2k$ bei Verwendung der expliziten Mittelpunktsregel. Die Approximation $T_{k,k-1}$ besitzt die Ordnung $2k - 2$. Für die Schrittweitenkontrolle benutzt man die Näherung

$$\epsilon_k = ||T_{k,k-1} - T_{k,k}|| \quad , \tag{1.20}$$

wobei analog zu (1.18) die Maximumsnorm verwendet werden kann. Analog (1.14) ergibt sich daraus der Schrittweitenvorschlag

$$h^{(k)} = h_\kappa \cdot \alpha \left(\frac{1}{\epsilon_k} \right)^{1/(2k-1)} \quad . \tag{1.21}$$

Im Gegensatz zu eingebetteten Runge-Kutta-Verfahren liefert ein Extrapolationstableau eine ganze Reihe von Näherungslösungen verschiedener Ordnung für $y(t_\kappa + h_\kappa)$, so daß Extrapolationsverfahren außer einer Schrittweitensteuerung auch eine Ordnungssteuerung erlauben. Eine optimale praktische Implementierung, die Schrittweiten- und Ordnungssteuerung miteinander kombinieren muß, ist deshalb um einiges komplizierter als für Runge-Kutta-Verfahren, die eine feste Ordnung besitzen. Die ersten Codes wurden von Bulirsch u. Stoer (1966) und deren Studenten entwickelt. Ein sehr erfolgreicher Extrapolationscode (DIFEX1) stammt von Deuflhard (1983) und seinen Mitarbeitern. Ein weiterer bekannter Code (ODEX) stammt von Hairer u. a. (2000).

Zur Durchführung der Ordnungssteuerung mißt man den Arbeitsaufwand für die Berechnung von $T_{k,1}$ bezogen auf die Grundschrittweite

$$W_k = \frac{A_k}{h^{(k)}} \quad , \tag{1.22}$$

wobei A_k die Anzahl der zur Berechnung von $T_{k,1}$ benötigten Funktionsauswertungen angibt. Die Idee ist nun, einen Spaltenindex k zu bestimmen, für den W_k minimal wird. Für weitere algorithmische Details sei an dieser Stelle allerdings auf die Originalarbeiten von Bulirsch u. Stoer (1966) und Deuflhard (1983, 1985) oder die Lehrbücher von Hairer u. a. (2000) und Strehmel u. Weiner (1995) verwiesen.

Schrittweitensteuerung als Regler. Schrittweitenvorschläge, die analog (1.14) bzw. (1.21) bestimmt werden, können zu Oszillationen im Bereich einer maximalen stabilen Schrittweite führen (siehe z. B. Hairer u. Wanner 2002). Dies kann sich sogar negativ auf die Glattheit der berechneten Lösungsfunktion auswirken. Hall u. Higham (1988) schlagen deshalb die Konstruktion neuer Lösungsverfahren vor, die mit der herkömmlichen Methode zur Schrittweitensteuerung besser zusammenarbeiten.

Ein alternativer und vielversprechenderer Ansatz besteht darin, die Schrittweitensteuerung zu verbessern, indem man die Anpassung der Schrittweite in Abhängigkeit von der Größe des lokalen Fehlers als regelungstechnischen Prozeß auffaßt. Dieser Ansatz wurde erstmalig von Gustafsson u. a. (1988) verfolgt,

die erkannten, daß sich die herkömmliche, aus der asymptotischen Entwicklung des lokalen Fehlers hergeleitete Vorschrift zur Berechnung der neuen Schrittweite als I-Regler (Integral-Regler) interpretieren läßt. Dieser Erkenntnis folgend, entwickelten sie eine verbesserte Schrittweitensteuerung durch Hinzufügen eines proportionalen Regelverhaltens, d. h. die Erweiterung des I-Reglers zu einem PI-Regler, wie er durch

$$h_{\text{neu}} = \left(\frac{\varepsilon}{\hat{r}_{\kappa+1}}\right)^{k_I} \left(\frac{\hat{r}_{\kappa}}{\hat{r}_{\kappa+1}}\right)^{k_P} h_{\kappa} \tag{1.23}$$

gegeben ist. Dabei ist $\varepsilon = \theta \cdot \text{TOL}$ die vom Nutzer vorgegebene Fehlertoleranz multipliziert mit einem Sicherheitsfaktor θ, und $\hat{r}_{\kappa} = ||\hat{l}_{\kappa}||$ bzw. $\hat{r}_{\kappa} = ||\hat{l}_{\kappa}/h_{\kappa}||$ ist die Norm des Näherungswertes $\hat{l}$ für den lokalen Fehler im entsprechenden Schritt (gegebenenfalls normiert bezüglich der Schrittweite). k_I und k_P sind Parameter des Reglers und stellen den sogenannten Integral- bzw. Proportionalteil dar.

Der Artikel von Söderlind (2002) faßt die bis zu diesem Zeitpunkt gewonnenen Erkenntnisse und Erfahrungen mit der Realisierung der Schrittweitensteuerung als PI-Regler zusammen. Söderlind (2003) reinterpretiert diese aus dem Blickwinkel der Filtertheorie, wodurch eine weitere Glättung durch das Herausfiltern hoher Frequenzen, d. h. schneller Änderungen der Schrittweite, möglich wird. Die Auswirkungen der Schrittweitenkontrolle auf die Stabilität untersuchen Söderlind u. Wang (2006).

1.1.2 Mehrschrittverfahren

Im Gegensatz zu Einschrittverfahren verwendet ein Mehrschrittverfahren (MSV) zur Berechnung eines neuen Approximationswertes mehrere zuvor berechnete Approximationswerte der gesuchten Lösungsfunktion. Werden r Näherungswerte verwendet ($r > 1$), so spricht man von einem r-Schrittverfahren. Die gebräuchlichsten r-Schrittverfahren haben auf einem äquidistanten Gitter folgende Gestalt:

$$\eta_{\kappa} + a_{r-1}\eta_{\kappa-1} + \cdots + a_0\eta_{\kappa-r} = h\,\mathbf{F}(t_{\kappa}; \eta_{\kappa}, \eta_{\kappa-1}, \ldots, \eta_{\kappa-r}; h; \mathbf{f}) \ . \tag{1.24}$$

Zur Durchführung solcher Verfahren werden r Startwerte $\eta_0, \eta_1, \ldots, \eta_{r-1}$ benötigt, die man z. B. mit Hilfe eines Einschrittverfahrens oder auf andere Weise bestimmen muß.

Lineare Mehrschrittverfahren

Bei vielen in der Praxis verwendeten Mehrschrittverfahren hängt die Funktion $\mathbf{F}$ wie folgt linear von $\mathbf{f}$ ab:

$$\mathbf{F}(t_{\kappa}; \eta_{\kappa}, \eta_{\kappa-1}, \ldots, \eta_{\kappa-r}; h; \mathbf{f}) \equiv b_r\mathbf{f}(t_{\kappa}, \eta_{\kappa}) + \cdots + b_0\mathbf{f}(t_{\kappa-r}, \eta_{\kappa-r}) \ . \tag{1.25}$$

Diese Verfahren bezeichnet man als *lineare Mehrschrittverfahren*. Sie sind durch die Koeffizienten $a_0, \ldots, a_{r-1}$ und $b_0, \ldots, b_r$ bestimmt. Gilt $b_r \neq 0$, spricht man von einem *impliziten* Verfahren; ist $b_r = 0$, liegt ein *explizites* Verfahren vor.

Für die Lösung nichtsteifer Anfangswertprobleme sind u. a. folgende lineare Mehrschrittverfahren in Gebrauch:

- explizite Adams-Verfahren (auch Adams-Bashford-Verfahren genannt)

$$\eta_{\kappa} = \eta_{\kappa-1} + h\sum_{l=1}^{r} b_{r-l}\mathbf{f}(t_{\kappa-l}, \eta_{\kappa-l}) \ ,$$

- implizite Adams-Verfahren (auch Adams-Moulton-Verfahren genannt)

$$\eta_{\kappa} = \eta_{\kappa-1} + h\sum_{l=0}^{r} b_{r-l}\mathbf{f}(t_{\kappa-l}, \eta_{\kappa-l}) \ ,$$

- Nyström-Verfahren

$$\eta_{\kappa} = \eta_{\kappa-2} + h\sum_{l=1}^{r} b_{r-l}\mathbf{f}(t_{\kappa-l}, \eta_{\kappa-l}) \ ,$$

- und Milne-Simpson-Verfahren

$$\eta_\kappa = \eta_{\kappa-2} + h \sum_{l=0}^{r} b_{r-l} \mathbf{f}(t_{\kappa-l}, \eta_{\kappa-l}) \ .$$

Implementierungen linearer Mehrschrittverfahren für nichtsteife Systeme erfolgen üblicherweise in Form von *Prädiktor-Korrektor-Prozessen*. Dazu verwendet man ein implizites lineares Mehrschrittverfahren (Korrektor), das effizient mit Hilfe einer Fixpunktiteration gelöst wird. Den Startwert für diese Iteration erhält man durch ein explizites lineares Mehrschrittverfahren (Prädiktor). Bei praktischen Implementierungen wird die Iteration nicht bis zur Konvergenz wiederholt, sondern es wird in der Regel nur eine feste Anzahl von Iterationen verwendet. Für eine ausführliche Darstellung siehe z. B. Strehmel u. Weiner (1995).

Die wichtigsten linearen Mehrschrittverfahren zur Lösung von steifen Anfangswertproblemen sind die von Curtis u. Hirschfelder (1952) eingeführten BDF-Verfahren (englisch: *backward differentiation formulas*), deren Verfahrensvorschrift lautet:

$$\sum_{l=1}^{r} \frac{1}{l} \nabla^l \eta_\kappa = h \, \mathbf{f}(t_\kappa, \eta_\kappa) \ .$$

Der Operator ∇^l bezeichnet dabei die *Rückwärtsdifferenzen*

$$\nabla^0 \eta_i = \eta_i \ , \quad \nabla^l \eta_i = \nabla^{l-1} \eta_i - \nabla^{l-1} \eta_{i-1} \quad \text{für} \quad l = 1, 2, \ldots \ .$$

Diese Verfahrensklasse zeichnet sich durch sehr gute Stabilitätseigenschaften bei gleichzeitig hoher Ordnung aus.

Allgemeine lineare Methoden

Die Verallgemeinerung von Runge-Kutta-Verfahren und linearen Mehrschrittverfahren führt zu Verfahren, die als *allgemeine linearen Methoden* bezeichnet werden. Sie wurden von Burrage u. Butcher (1980) eingeführt. Hier hängen im Schritt n die ausgehenden Approximationen $\eta_1^{(n+1)}, \eta_2^{(n+1)}, \ldots, \eta_r^{(n+1)}$ von den eingehenden Approximationen $\eta_1^{(n)}, \eta_2^{(n)}, \ldots, \eta_r^{(n)}$ wie folgt ab:

$$\mathbf{v}_i = \sum_{j=1}^{r} a_{ij}^{(1)} \eta_j^{(n)} + h \sum_{j=1}^{s} b_{ij}^{(1)} \mathbf{f}\left(t_n + c_j h, \mathbf{v}_j\right) \quad \text{für} \quad i = 1, \ldots, s \ ,$$

$$\eta_i^{(n+1)} = \sum_{j=1}^{r} a_{ij}^{(2)} \eta_j^{(n)} + h \sum_{j=1}^{s} b_{ij}^{(2)} \mathbf{f}\left(t_n + c_j h, \mathbf{v}_j\right) \quad \text{für} \quad i = 1, \ldots, r \ .$$

Die Stufen $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_s$ sind *interne Stufen*, die nur innerhalb des aktuellen Schrittes verwendet werden. $\eta_1^{(n)}, \eta_2^{(n)}, \ldots, \eta_r^{(n)}$ werden auch als *externe Stufen* bezeichnet, da sie Informationen aus dem vorhergehenden Schritt in den aktuellen Schritt transportieren.

Weiterführende Informationen zu allgemeinen linearen Methoden sind in der von Butcher (2003) veröffentlichten Monographie zu finden. Zu den aktuellen Arbeiten auf diesem Gebiet gehören Butcher (2001) und Wright (2002).

Schrittweiten- und Ordnungssteuerung

Wie für Einschritt- und Extrapolationsverfahren ist auch für effiziente Implementierungen von Mehrschrittverfahren eine dynamische Anpassung von Schrittweite und Ordnung erforderlich. Es ergibt sich dabei allerdings die Schwierigkeit, daß Approximationswerte aus vorhergehenden Zeitschritten verwendet werden müssen, die Verfahrensvorschriften aber zunächst nur auf äquidistanten Gittern definiert wurden.

Um dennoch eine Anpassung der Schrittweite zu ermöglichen, kann man sich die Werte an den erforderlichen Zwischenpunkten mittels Polynominterpolation beschaffen. Mit Hilfe der Newtonschen Interpolationsformel lassen sich aber auch direkt Verfahrensvorschriften für variable Gitter, z. B. für die Adams-Verfahren (Ceschino 1961; Forrington 1961; Krogh 1969) und analog für die BDF-Verfahren, angeben. Der

Rechenaufwand für die Bestimmung des variablen Gitters ist hierbei allerdings recht hoch. Dafür ermöglicht dieses Vorgehen aber eine sehr einfache Ordnungssteuerung.

Eine alternative Möglichkeit zur Schrittweitenanpassung ist die Verwendung der *Nordsieck-Darstellung* (Nordsieck 1962). Hierbei wird das ursprüngliche r-Schrittverfahren in ein Verfahren transformiert, welches formal nur den letzten Zeitschritt sieht. Dadurch wird eine bequeme Schrittweitenänderung bei konstanter Ordnung möglich, der Wechsel der Ordnung führt allerdings zu umfangreichen Nebenrechnungen.

1.1.3 Waveform-Relaxationsverfahren

Waveform-Relaxationsverfahren (WR-Verfahren) wurden in den Arbeiten von Lelarasmee (1982) und Lelarasmee u. a. (1982) zur Lösung von Problemen, die bei der Simulation von VLSI-Schaltkreisen (von englisch: *very large scale integrated circuits*) entstehen, eingeführt. Hierbei entstehen sehr große, steife Systeme, die am besten durch eine Unterteilung sowie eine separate Behandlung der Teilsysteme gelöst werden können.

Die grundlegende Idee der Waveform-Relaxationsverfahren besteht darin, beginnend mit einer näherungsweise bestimmten Lösungsfunktion $\mathbf{y}^{(0)}(t)$, eine Folge von gewöhnlichen Differentialgleichungen für die Lösungsfunktionen $\mathbf{y}^{(1)}(t), \ldots, \mathbf{y}^{(p)}(t)$ zu lösen, mit der Hoffnung, daß diese Folge gegen die exakte Lösungsfunktion $\mathbf{y}(t)$ konvergiert. Ein Beispiel für diesen Ansatz ist die Picard-Methode. Hier wird die Folge gewöhnlicher Differentialgleichungen

$$\mathbf{y}^{(k+1)}(t) = \mathbf{f}(t, \mathbf{y}^{(k)}(t)) \ , \quad \mathbf{y}^{(k+1)}(0) = \mathbf{y}_0 \ , \quad t \in [t_0, t_e]$$

berechnet, deren Lösung sich zu

$$\mathbf{y}^{(k+1)}(t) = \mathbf{y}_0 + \int_{t_0}^{t} \mathbf{f}(\xi, \mathbf{y}^{(k)}(\xi))\, d\xi$$

ergibt. Der größte Nachteil der Picard-Iteration ist die langsame Konvergenz, weshalb verschiedene Techniken zur Verbesserung der Konvergenzgeschwindigkeit vorgeschlagen wurden (siehe z. B. Skeel 1989).

Allgemeinere Waveform-Verfahren nutzen zusätzlich bereits berechnete Informationen aus der aktuellen Iteration, um auf diese Weise bessere Konvergenzeigenschaften zu erzielen. Das heißt, man setzt

$$\begin{aligned}
\mathbf{z}^{(k+1)\prime}(t) &= \mathbf{G}(t, \mathbf{z}^{(k+1)}(t), \mathbf{y}^{(k+1)}(t), \mathbf{y}^{(k)}(t)) \ , \qquad \mathbf{z}^{(k+1)}(t_0) = \mathbf{y}_0 \ , \\
\mathbf{y}^{(0)}(t_0) &= \mathbf{y}_0 \ , \\
\mathbf{y}^{(k+1)}(t) &= \mathbf{g}(t, \mathbf{z}^{(k+1)}(t), \mathbf{y}^{(k)}(t)) \ ,
\end{aligned}$$

wobei die Unterteilungsfunktionen $\mathbf{G}$ und $\mathbf{g}$ die Bedingungen

$$\begin{aligned}
\mathbf{G}(t, \mathbf{y}, \mathbf{y}, \mathbf{y}) = \mathbf{f}(t, \mathbf{y}) \ , \quad \mathbf{G} : [t_0, t_e] \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n \ , \\
\mathbf{g}(t, \mathbf{y}, \mathbf{y}) = \mathbf{f}(t, \mathbf{y}) \ , \quad \mathbf{g} : [t_0, t_e] \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n \ ,
\end{aligned}$$

erfüllen müssen. Abhängig von der verwendeten Unterteilung unterscheidet man in Analogie zu Lösungsverfahren für lineare Gleichungssysteme u. a. zwischen

- Jacobi-WR-Verfahren:

$$y_i^{(k+1)\prime} = f_i(t, y_0^{(k)}, \ldots, y_{i-1}^{(k)}, y_i^{(k+1)}, y_{i+1}^{(k)}, \ldots, y_n^{(k)}) \ , \qquad i = 1, \ldots, n \ ,$$

- Gauß-Seidel-WR-Verfahren:

$$y_i^{(k+1)\prime} = f_i(t, y_0^{(k+1)}, \ldots, y_{i-1}^{(k+1)}, y_i^{(k+1)}, y_{i+1}^{(k)}, \ldots, y_n^{(k)}) \ , \qquad i = 1, \ldots, n \ ,$$

- und SOR-WR-Verfahren:

$$\begin{aligned}
z_i^{(k+1)\prime} &= f_i(t, y_0^{(k+1)}, \ldots, y_{i-1}^{(k+1)}, z_i^{(k+1)}, y_{i+1}^{(k)}, \ldots, y_n^{(k)}) \ , \qquad i = 1, \ldots, n \ , \\
y_i^{(k+1)} &= w z_i^{(k+1)} + (1 - w) y_i^{(k)} \ , \qquad i = 1, \ldots, n \ .
\end{aligned}$$

Die von Skeel (1989) vorgeschlagenen Techniken lassen sich auch auf die verallgemeinerten WR-Verfahren übertragen (Nevanlinna 1990; Lubich 1992). Darüber hinaus ist eine Verbesserung der Konvergenz durch Multisplitting, d. h. durch eine durch mehrfache Unterteilung erzeugte Überlappung, möglich (Jeltsch u. Pohl 1991; Pohl 1992; Frommer u. Pohl 1995).

Der besondere Vorteil der WR-Verfahren besteht darin, daß eine Entkoppelung von Teilsystemen möglich ist, so daß Teilsysteme mit unterschiedlichen Eigenschaften unabhängig voneinander behandelt werden können. Dadurch ist es möglich, für jedes Teilsystem ein passendes Integrationsverfahren auszuwählen, und die Teilsysteme können mit unterschiedlichen Schrittweiten integriert werden. Neben der Anpassung der zeitlichen Diskretisierung ermöglichen Multigrid-Waveform-Relaxationsverfahren auch eine Anpassung des räumlichen Gitters bei der Lösung semi-diskretisierter partieller Differentialgleichungen (abgekürzt: *PDEs* für englisch: *partial differential equations*) (siehe Lubich u. Ostermann 1987; Vandewalle 1993; van Lent u. Vandewalle 2002). Als Basisverfahren können z. B. lineare Mehrschrittverfahren oder Runge-Kutta-Verfahren eingesetzt werden. Für einen ausführlichen Überblick siehe Burrage (1995).

1.2 Parallele Lösungsverfahren

Bei einer parallelen Implementierung von Lösungsverfahren für gewöhnliche Differentialgleichungssysteme lassen sich verschiedene Parallelitätsarten unterscheiden. Nach Gear (1987, 1988) unterscheidet man aus Sicht der Lösungsverfahren folgende Parallelitätsarten:

- ***Parallelität bezüglich des Systems:*** Verschiedene Komponenten des gewöhnlichen Differentialgleichungssystems werden parallel behandelt.
- ***Parallelität bezüglich der Methode:*** Unabhängige Berechnungen innerhalb eines Zeitschrittes werden parallel ausgeführt.
- ***Parallelität bezüglich der Zeitschritte:*** Mehrere Zeitschritte werden parallel ausgeführt.

Aus Sicht der Algorithmen, die diese Lösungsverfahren realisieren, unterscheidet man:

- ***Instruktionsparallelität:*** Mehrerer Maschineninstruktionen werden durch unterschiedliche Funktionseinheiten eines einzelnen Prozessors gleichzeitig ausgeführt.
- ***Datenparallelität:*** Durch einen oder mehrere Prozessoren werden gleiche Instruktionen auf unterschiedliche Daten parallel angewendet.
- ***Taskparallelität:*** Unabhängige Instruktionsfolgen (Tasks) werden durch einen oder mehrere Prozessoren parallel ausgeführt, wobei Tasks selbst ebenfalls parallelisierbar sein können.

Die parallele Implementierung von Verfahren zur Lösung von gewöhnlichen Differentialgleichungen wurde lange Zeit vernachlässigt, da die bekannten Verfahren als inhärent sequentiell galten. In der jüngeren Vergangenheit wurden jedoch von Mathematikern mehrere Verfahren zur Lösung gewöhnlicher Differentialgleichungssysteme vorgeschlagen, die gut für eine parallele Implementierung geeignet sind, da sie ein größeres Potential an Parallelität besitzen als herkömmliche Methoden.

Die verschiedenen Verfahren wurden meist von Mathematikern mit dem Ziel vorgeschlagen, neue numerische Verfahren zu entwerfen, die voneinander unabhängige Berechnungen aufweisen und deren numerische Eigenschaften mit den numerischen Eigenschaften der bekannten Standardverfahren, wie sie z. B. von Hairer u. a. (2000) und Hairer u. Wanner (2002) beschrieben werden, vergleichbar sind. Die entsprechenden Veröffentlichungen enthalten meist eine genaue Analyse der numerischen Eigenschaften (Konvergenz, Stabilität) der Verfahren; üblicherweise wird aber ein paralleler Algorithmus, der eine genaue Abbildung der durchzuführenden Berechnungsschritte auf parallel arbeitende Berechnungseinheiten angibt, nur sehr vage beschrieben oder fehlt ganz. Oft wird nur ein zusätzlicher Grad an Parallelität identifiziert, Wechselwirkungen mit anderen Parallelitätsquellen werden aber nicht diskutiert.

Im folgenden soll ein kurzer Überblick über die wichtigsten parallelen Lösungsverfahren gegeben werden. Eine detaillierte Einführung in dieses Thema gibt Burrage (1995).

1.2.1 Runge-Kutta-Verfahren

Die Möglichkeiten zur parallelen Ausführung von Runge-Kutta-Verfahren werden durch die Berechnungsvorschrift (1.9) bestimmt. Da, wie bei allen Ein- und Mehrschrittverfahren, Abhängigkeiten zwischen aufeinanderfolgenden Zeitschritten bestehen, ist eine Parallelisierung bezüglich der Zeitschritte nur in Ausnahmenfällen durch Überlappung von wenigen direkt benachbarten Zeitschritten möglich. Eine Anpassung der Schrittweite nach jedem Zeitschritt erschwert dies zusätzlich. Eine Parallelisierung bezüglich des Differentialgleichungssystems und bezüglich der Methode ist jedoch prinzipiell möglich.

Explizite Runge-Kutta-Verfahren

Bei expliziten Runge-Kutta-Verfahren hängen die Stufen (1.9a) sequentiell voneinander ab, so daß eine parallele Berechnung der Stufen im allgemeinen Fall nicht möglich ist. Die Möglichkeit einer parallelen Ausführung besteht jedoch für spezielle Verfahren, deren Verfahrensmatrix besondere Eigenschaften aufweist.

Einer der ersten Artikel über parallele Lösungsverfahren für gewöhnliche Differentialgleichungssysteme von Miranker u. Liniger (1967) enthält im Anhang die Beschreibung eines parallelen expliziten Runge-Kutta-Verfahrens der Ordnung 3. Hier wird eine parallele Berechnung der Stufen dadurch ermöglicht, daß das Verfahren eingebettete Lösungen der Ordnung 1 und 2 enthält und zusätzlich eine konstante Schrittweite vorausgesetzt wird. Die eingebetteten Lösungen und die zugehörigen Stufen werden auf unterschiedlichen Prozessoren berechnet, wobei die Berechnung einer Lösung der Ordnung p der Berechnung der Lösung der Ordnung $p+1$ um einen Zeitschritt vorausläuft. Zur Berechnung der Lösung der Ordnung $p+1$ muß deshalb nur noch die Stufe $p+1$ berechnet werden, da die Stufen $i = 1, \ldots, p$ bereits auf Prozessor i zur Bestimmung der Lösung der Ordnung i berechnet wurden. Khalaf u. Hutchinson (1992) zeigen jedoch, daß die Konstruktion derartiger Verfahren für eine Ordnung größer als 3 nicht durchführbar ist. Dies bedeutet insbesondere, daß diese Art der Parallelisierung auf maximal drei Prozessoren beschränkt ist. Darüber hinaus besitzt das von Miranker u. Liniger (1967) vorgeschlagene Verfahren schlechte Stabilitätseigenschaften, wie die Autoren selbst einräumen.

Eine detaillierte Analyse des Parallelitätspotentials von Runge-Kutta-Verfahren bezüglich der parallelen Berechnung der Stufen wurde von Iserles u. Nørsett (1990) und Jackson u. Nørsett (1995) durchgeführt. Sie fanden heraus, daß eine Parallelisierung expliziter Runge-Kutta-Verfahren nur begrenzt möglich ist:

Satz 1.1. *Für ein explizites Runge-Kutta-Verfahren mit σ sequentiellen Stufen unterliegt die Ordnung p der Bedingung*

$$p \leq \sigma$$

für jede beliebige Anzahl verfügbarer Prozessoren.

Unter der Anzahl der sequentiellen Stufen σ versteht man dabei die Länge des längsten Pfades im *Produktionsgraphen* des Verfahrens, der die Abhängigkeiten zwischen den Stufen beschreibt. Für einen Beweis siehe die Originalarbeiten von Iserles u. Nørsett (1990) und Jackson u. Nørsett (1995) oder Hairer u. a. (2000). Verfahren mit $p = \sigma$ bezeichnet man als p-optimal. Für die Parallelisierung eines p-optimalen s-stufigen expliziten Runge-Kutta-Verfahrens bedeutet Satz 1.1, daß ein solches Verfahren maximal $s - p$ Stufen besitzt, die für eine parallele Ausführung in Betracht kommen. Die Verteilung der parallel ausführbaren Stufen auf verschiedene Prozessoren kann mittels der von Iserles u. Nørsett (1990) vorgeschlagenen *Digraphmethode* durchgeführt werden. Diesen Ansatz verfolgen auch Petcu (1999) und Petcu u. Dragan (2000), deren Software *EpODE* (für englisch: *expert system for ordinary differential equations*) nach Vorgabe eines Runge-Kutta-Verfahrens eine automatische Verteilung der Stufenvektorberechnungen durchführt.

Eine Untersuchung des Parallelitätspotentials eingebetteter Runge-Kutta-Verfahren bezüglich des Differentialgleichungssystems wurde von Rauber u. Rünger (1999b) durchgeführt. Dazu wurden Implementierungen für gemeinsamen und für verteilten Speicher betrachtet und eine theoretische Abschätzung der Kommunikations- und Berechnungskosten durchgeführt. Laufzeitexperimente wurden auf zwei Parallelrechnern mit verteiltem Speicher, einer Intel Paragon und einer IBM SP2, durchgeführt. Gute Speedup-Werte konnten dabei jedoch nur beobachtet werden, wenn die Zeit für die Auswertung der rechten Seite des Differentialgleichungssystems ausreichend hoch war, um die Kosten der aufgrund des verteilten Adreßraumes auf beiden Parallelrechnern sehr teuren Kommunikationsoperationen auszugleichen.

Implizite Runge-Kutta-Verfahren

Für implizite Runge-Kutta-Verfahren ergibt die Berechnungsvorschrift (1.9a) eine implizite Beziehung der Stufenvektoren, so daß für ein allgemeines s-stufiges Verfahren und ein Differentialgleichungssystem der Dimension n in jedem Zeitschritt die Lösung eines nichtlinearen Gleichungssystems der Dimension $s \cdot n$ erforderlich ist. Im Falle einer vollbesetzten Verfahrensmatrix kann Parallelität daher im wesentlichen nur bei der Lösung des nichtlinearen Gleichungssystems ausgenutzt werden.

Erfüllt jedoch die Struktur der Verfahrensmatrix A bestimmte Voraussetzungen, ist eine parallele Berechnung der Stufen prinzipiell möglich. Einfachstes Beispiel sind die sogenannten streng diagonal-impliziten Runge-Kutta-Verfahren, die durch $a_{li} = 0$ für $l \neq i$ gekennzeichnet sind. Da in diesem Fall jeder Stufenvektor nur von sich selbst abhängig ist, können alle Stufenvektoren gleichzeitig berechnet werden. Das bedeutet insbesondere auch, daß anstelle des nichtlinearen Gleichungssystems der Dimension $s \cdot n$ jetzt s unabhängige nichtlineare Gleichungssysteme der Dimension n gelöst werden müssen, was gleichzeitig zu einer Reduzierung des Berechnungsaufwandes führt. Jackson u. Nørsett (1995) zeigen jedoch, daß derartige Verfahren für allgemeine Probleme nur eine maximale Ordnung von 2 erreichen können.

Bessere Ordnungseigenschaften bieten dagegen die parallelen mehrfach-impliziten Runge-Kutta-Verfahren (PMIRK-Verfahren), die ebenfalls eine Entkoppelung in s unabhängige nichtlineare Gleichungssysteme der Dimension n ermöglichen (vgl. Burrage 1995). Dies wird durch die Voraussetzung möglich, daß die Verfahrensmatrix A durch eine Ähnlichkeitstransformation in eine Diagonalmatrix D überführt werden kann, d. h., es gilt:

$$A = T^{-1}DT \ , \qquad D = \mathrm{diag}(\lambda_1, \ldots, \lambda_s) \ .$$

Vergleicht man jedoch die für eine parallele Implementierung eines PMIRK-Verfahrens erforderliche Zahl von Operationen mit der erforderlichen Anzahl von Operationen für ein effizientes sequentielles SIRK-Verfahren, so resultieren aus der Parallelität keine nennenswerten Vorteile.

Iterierte Runge-Kutta-Verfahren

Eine Verfahrensklasse mit vielversprechendem Parallelitätspotential sind die *Prädiktor-Korrektor-Verfahren*. Prädiktor-Korrektor-Verfahren versuchen die Vorteile der einfachen Berechnungsstruktur expliziter Verfahren mit der verbesserten Stabilität und Genauigkeit impliziter Verfahren zu kombinieren. Wie bereits in Abschnitt 1.1.2 im Zusammenhang mit linearen Mehrschrittverfahren angesprochen, geht man dazu von einem impliziten Basisverfahren aus und verwendet einen iterativen Ansatz zur Auflösung der nichtlinearen Gleichungssysteme. Als Startwert der Iteration verwendet man einen Näherungswert der Lösung, der mit Hilfe eines expliziten Verfahrens ermittelt wird. Dieses Verfahren wird daher als *Prädiktorverfahren* bezeichnet. Anschließend wird die Näherungslösung in mehreren Iterationsschritten mit Hilfe des impliziten Basisverfahrens „korrigiert“, weshalb das Basisverfahren auch *Korrektorverfahren* genannt wird.

Handelt es sich bei dem Basisverfahren um ein Runge-Kutta-Verfahren, so spricht man von einem *iterierten Runge-Kutta-Verfahren*. Das einfachste und bekannteste Verfahren dieser Art, das daher auch oft als *das* iterierte Runge-Kutta-Verfahren bezeichnet wird, verwendet den sogenannten „trivialen“ Prädiktor

$$\mathbf{Y}_l^{(0)} = \eta_\kappa \quad \text{für} \quad l = 1, \ldots, s \tag{1.26}$$

als Startwert für eine Fixpunktiteration mit einer festen Anzahl m von Iterationen:

$$\mathbf{Y}_l^{(j)} = \eta_\kappa + h_\kappa \sum_{i=1}^{s} a_{li} \mathbf{f}(t_\kappa + c_l h_\kappa, \mathbf{Y}_i^{(j-1)}) \quad \text{für} \quad l = 1, \ldots, s \quad \text{und} \quad j = 1, \ldots, m \ . \tag{1.27}$$

Die neue Approximation für die Lösungsfunktion an der Stelle $t_{\kappa+1}$ erhält man anschließend aus

$$\eta_{\kappa+1} = \eta_\kappa + h_\kappa \sum_{l=1}^{s} b_l \mathbf{f}(t_\kappa + c_l h_\kappa, \mathbf{Y}_l^{(m)}) \ . \tag{1.28}$$

Dieses iterierte Verfahren geht zurück auf die Arbeiten von Nørsett u. Simonsen (1989) und van der Houwen u. Sommeijer (1990a). Es läßt sich als explizites Verfahren mit $s(m+1)$ Stufen auffassen, dessen

Verfahrensmatrix gegeben ist durch:

$$\begin{array}{c|ccccc} 0 & 0 & & & & \\ \mathbf{c} & A & 0 & & & \\ \mathbf{c} & 0 & A & 0 & & \\ \vdots & \vdots & & \ddots & \ddots & \\ \mathbf{c} & 0 & \dots & 0 & A & 0 \\ \hline & 0 & \dots & 0 & 0 & \mathbf{b}^T \end{array} \tag{1.29}$$

Bezeichnet man die Ordnung des Basisverfahrens mit p, so beträgt die Ordnung eines solchen Verfahrens $p^* = \min\{p, m+1\}$. Die Anzahl der Korrekturen m wird dementsprechend in der Regel auf $m = p-1$ gesetzt, um eine höchstmögliche Ordnung bei gleichzeitig möglichst geringer Iterationszahl zu erreichen.

Der Vorteil eines solchen Verfahrens besteht darin, daß innerhalb jedes Iterationsschrittes j die Vektoren $\mathbf{Y}_1^{(j)}, \dots, \mathbf{Y}_s^{(j)}$ unabhängig voneinander sind und daher parallel berechnet werden können. Man erhält somit ein Lösungsverfahren der Ordnung $p^* = p$ mit lediglich p sequentiellen Stufen. Dies bedeutet insbesondere, daß das Verfahren optimal in bezug auf Satz 1.1 ist. Im Vergleich dazu wächst die für die Berechnung eines Zeitschrittes eines sequentiellen expliziten Verfahrens erforderliche Stufenzahl schneller als linear im Verhältnis zur Ordnung.

Eine ausführliche Untersuchung des Parallelitätspotentials iterierter Runge-Kutta-Verfahren, die den trivialen Prädiktor sowie eine feste Anzahl von Iterationen eines impliziten Runge-Kutta-Korrektors verwenden, wurde von Rauber u. Rünger (1996, 1999b) durchgeführt. Dazu wurden verschiedene Implementierungen für Parallelrechner mit verteiltem Speicher erstellt und untersucht, die sowohl Parallelität bezüglich der Methode als auch Parallelität bezüglich des Differentialgleichungssystems ausnutzen, jedoch jeweils unterschiedliche Datenverteilungen verwenden. Für diese Implementierungen wurde eine theoretische Analyse der Berechnungs- und der Kommunikationszeit durchgeführt, die eine sehr genaue Vorhersage der Laufzeit ermöglicht. Eine Untersuchung des realen Laufzeitverhaltens erfolgte auf Parallelrechnern vom Typ Intel iPSC/860, Intel Paragon und IBM SP2. Zufriedenstellende Speedup-Werte konnten jedoch nur gemessen werden, wenn die Funktionsauswertungszeit des zu lösenden Differentialgleichungssystems sehr hoch war.

Neben den eben besprochenen Verfahren existieren eine Reihe weiterer iterierter RK-Verfahren. (Parallele) diagonal-implizite iterierte RK-Verfahren (DIIRK- bzw. PDIRK-Verfahren) entstehen durch eine Variation des Korrektorschrittes (1.27), die zu einem impliziten Verfahren führt, das auch zur Lösung von steifen Differentialgleichungen geeignet ist, das aber ebenfalls einen höheren Grad an Parallelität als andere implizite Verfahren besitzt (van der Houwen u. a. 1992). Da diese Veränderung von (1.27) dazu führt, daß in jedem Iterationsschritt s implizite, in der Regel nichtlineare Gleichungssysteme gelöst werden müssen, ergibt sich jedoch eine wesentlich höhere Berechnungszeit. Eine detaillierte Analyse des Laufzeitverhaltens wurde von Rauber u. Rünger (1999a, c) durchgeführt.

Zur Verbesserung der Konvergenzeigenschaften der DIIRK-Verfahren wurden weiterhin parallele triangular-implizite iterierte Verfahren (PTIRK-Verfahren) vorgeschlagen (van der Houwen u. de Swart 1997). Diese Verfahren besitzen einen etwas geringeren Grad an Parallelität und erfordern einen höheren Berechnungsaufwand pro Iterationsschritt, sie benötigen jedoch deutlich weniger Iterationsschritte um eine vorgegebene Genauigkeit zu erreichen und ermöglichen dadurch gleichzeitig eine Überlappung der Berechnung mehrerer Zeitschritte. Die Idee der PTIRK-Verfahren wurde von Burrage u. Suhartanto (1997a) für nichtsteife und ebenfalls von Burrage u. Suhartanto (1997b) für steife Probleme auf Mehrschritt-Runge-Kutta-Verfahren des Radau-Typs verallgemeinert.

1.2.2 Extrapolationsverfahren

Wie in Abschnitt 1.1.1 beschrieben, berechnen Extrapolationsverfahren für die gleiche Argumentstelle verschiedene Approximationen der gesuchten Lösungsfunktion, indem sie – ausgehend von der errechneten Lösung für die vorangehende Argumentstelle – verschiedene Schrittweiten verwenden. Durch Kombination dieser Approximationen entsteht eine Approximation höherer Konvergenzordnung. Wegen der Unabhängigkeit der Berechnungen der Approximationen für eine Argumentstelle mit verschiedenen Schrittweiten sind Extrapolationsverfahren mehrfach für eine parallele Ausführung diskutiert worden, u. a. von

Deuflhard (1985), Ehrig u. a. (1998), Hairer u. a. (2000), Kappeller u. a. (1996), Rauber u. Rünger (1997a) sowie van der Houwen u. Sommeijer (1990b).

1.2.3 Mehrschrittverfahren und allgemeine lineare Methoden

Neben den genannten Verfahren wurden viele weitere für eine parallele Ausführung vorgeschlagen, darunter auch viele Verfahren, die zur Klasse der Mehrschrittverfahren bzw. der allgemeinen linearen Methoden gehören. Unter anderem sind dies Variationen von RK-Verfahren, wie z. B. EPTRK- (Podhaisky u. a. 1999), EPTRKN- (Cong u. a. 2000) und SW-VerfahrenPPSW-Verfahren (Schmitt u. Weiner 2004; Weiner u. a. 2003), aber auch BDF-Verfahren (Frank u. van der Houwen 2001) und parallele Adams-Verfahren (van der Houwen u. Messina 1999). Burrage (1995) gibt einen ausführlichen Überblick.

1.2.4 Waveform-Relaxationsverfahren

Für eine parallele Implementierung haben die in Abschnitt 1.1.3 vorgestellten Waveform-Relaxationsverfahren den Vorteil eines hohen Grades an Parallelität, da Teilsysteme des gewöhnlichen Differentialgleichungssystems unabhängig voneinander berechnet werden können und für jedes Teilsystem wieder eine Berechnungsmethode mit geeignetem Grad an Parallelität verwendet werden kann. Parallele Implementierungen von Waveform-Relaxationsverfahren werden u. a. von Burrage (1995), Schaller (1998), van Lent u. Vandewalle (2002) sowie Gander u. a. (2003) beschrieben.

1.3 Bedeutung der Speicherzugriffslokalität

Eine wesentliche Charakteristik heutiger Rechnersysteme ist das Vorhandensein einer Speicherhierarchie, deren Einführung durch das Auseinanderdriften von Prozessorgeschwindigkeit und Speicherzugriffsgeschwindigkeit notwendig wurde (siehe Culler u. a. 1999). Das Anhalten dieses Trends führt zu immer tieferen Speicherhierarchien (Hennessy u. Patterson 2003).

Dies trifft nicht nur auf sequentielle Rechner zu. Moderne Parallelrechner, insbesondere Parallelrechner mit gemeinsamem Speicher, bei denen sich alle Prozessoren einen gemeinsamen Adreßraum teilen, weisen im Vergleich oft sogar eine noch tiefere Speicherhierarchie auf. Ein Beispiel hierfür sind die Parallelrechner der von SGI vertriebenen Altix-Reihe. Ein solches System wird z. Z. im Leibniz-Rechenzentrum in München installiert. Für die Endausbaustufe, die 2007 verfügbar sein wird, ist die Ausstattung mit 9.728 Itanium-Montecito-Prozessorkernen und 40 TB Hauptspeicher geplant. Die Hardwarearchitektur des Systems namens NUMAflex bildet den gesamten Hauptspeicher in einen gemeinsamen Adreßraum ab (SGI 2003, 2005). Die Basis dafür bildet ein routerbasiertes Verbindungsnetzwerk, das in einer sogenannten „Fat-Tree“-Topologie organisiert ist. Bezüglich der Latenzzeiten, die für Speicherzugriffe auf diesem System auftreten werden, lassen sich zur Zeit noch keine genauen Angaben machen, da sich die für die Endausbaustufe vorgesehenen Prozessoren noch in der Entwicklung befinden. Derzeit lieferbare Itanium-2-Prozessoren besitzen eine dreistufige Cache-Hierarchie mit minimalen Latenzzeiten von 1, 5 bzw. 14 Taktzyklen (Intel 2002, 2004). Legt man die z. Z. maximale verfügbare Taktfrequenz von 1,7 GHz zugrunde, entspricht dies Cache-Zugriffslatenzen von 0,6, 2,9 und 8,2 ns. Die Latenzzeiten für Hauptspeicherzugriffe auf nicht im Cache gespeicherte Daten hängen von einer Reihe von Faktoren ab. Der wichtigste Einflußfaktor ist die physikalische Position der angeforderten Speicheradresse, denn für den Zugriff auf Speichermodule, die nicht direkt an den anfordernden Prozessor angebunden sind, durchläuft die Speicheranfrage die Baumtopologie des Verbindungsnetzwerks. Für ein Altix-System mit 64 Prozessoren dauert ein Speicherzugriff daher etwa zwischen 140 und 515 ns. Auf Systemen mit größerer Prozessoranzahl wächst die maximale Speicherzugriffslatenz entsprechend an. Auf einem 512-Prozessor-System liegt sie beispielsweise bei ca. 650 ns.

Aufgrund der zunehmenden Tiefe der Speicherhierarchie, spielen für die Effizienz eines Programms die zeitlichen und räumlichen Lokalitätseigenschaften der von ihm verursachten Speicherzugriffe eine große Rolle. Insbesondere für parallele Programme ist die geeignete Anordnung der Speicherzugriffe der einzelnen Prozessoren ein wesentliches Entwurfskriterium (Culler u. a. 1999; Handy 1998). Dies ist leicht nachzuvollziehen, wenn man bedenkt, daß bei häufiger zeitlicher Wiederverwendung von Daten eine geringere Belastung des Verbindungsnetzwerks möglich wird, da eine wiederholte Übertragung von Daten durch

Zwischenspeicherung oft vermieden werden kann. Darüber hinaus können bei einer hohen räumlichen Lokalität die Daten in Form größerer Blöcke übertragen und verarbeitet werden. Dies ist besonders vorteilhaft, da die Speicherhierarchien modernen Rechnersysteme in der Regel für eine blockweise Datenübertragung optimiert sind (Meyer u. a. 2003).

Ansätze für eine effiziente Nutzung von Speicherhierarchien gibt es im Bereich des Algorithmenentwurfs und des Übersetzerbaus. Im Bereich des Algorithmenentwurfs wurden z. B. von LaMarca u. Ladner (1997) Untersuchungen und Experimente durchgeführt, die zeigen, daß Lokalitätsaspekte beim Entwurf von Algorithmen für eine Realisierung auf realen Rechnern eine große Rolle spielen, daß aber der Entwurf der Algorithmen wesentlich komplexer werden kann. Im Bereich des Übersetzerbaus wird eine effiziente Nutzung der Speicherhierarchie durch eine Umordnung der Anweisungen des Quellprogramms zu erreichen versucht (siehe Wolfe 1996), wobei sich die Untersuchungen meist auf rechenzeitintensive Schleifen konzentrieren. Arbeiten auf diesem Gebiet wurden u. a. von McKinley (1998), Kandemir u. a. (2000a, b) sowie Jin u. a. (2001) durchgeführt. Als eine der wichtigsten Techniken hat sich dabei Schleifen-*Tiling* erwiesen, das z. B. von Irigoin u. Triolet (1988), Coleman u. McKinley (1995), McKinley (1998), Abella u. a. (2002) und Drosinos u. a. (2004) untersucht wurde. Eine für die Abschätzung der Effekte von Schleifentransformationen wichtige analytische Modellierung wird von Ghosh u. a. (1999) beschrieben. Lindenmaier u. a. (2000) beschreiben, wie Performance-Counter-Informationen genutzt werden können, um durch Instruktionsparallelität Latenzzeiten bei Speicherzugriffen zu verstecken. Ein Überblick über verschiedene Optimierungstechniken zur effizienten Ausnutzung der Speicherhierarchie geben Kowarschik u. Weiß (2003). Diese Techniken ermöglichen u. a. die Beschleunigung von Lösungsverfahren für partielle Differentialgleichungen (Kowarschik u. a. 2002; Wilke u. a. 2003).

Im Rahmen der Erforschung parallelisierender Compiler werden Techniken entwickelt, die eine automatische Identifizierung feinkörniger Parallelität ermöglichen und auch auf eine verbesserte Ausnutzung der Speicherhierarchie zielen (Wolfe 1996; Allen u. Kennedy 2002; Hwang u. Saltz 2003). Diese Techniken beruhen auf einer Datenabhängigkeitsanalyse auf Programmebene. Sie sind deshalb zwar in der Lage, ein gegebenes Quellprogramm in ein effizientes Zielprogramm zu übertragen, sie sind jedoch darauf angewiesen, daß der Programmierer einen geeigneten, effizienten Algorithmus zur Lösung des betrachteten Problems als Quellprogramm zur Verfügung stellt.

1.4 Beitrag dieser Arbeit

Das Thema dieser Arbeit sind Implementierungsaspekte eingebetteter Runge-Kutta-Verfahren zur Lösung von Anfangswertproblemen gewöhnlicher Differentialgleichungssysteme, die auf das Erzielen einer möglichst geringen Laufzeit ausgerichtet sind und das Zielsystem möglichst effizient ausnutzen. Als Zielsysteme werden sequentielle, aber auch parallele Rechnersysteme sowohl mit verteiltem als auch mit gemeinsamem Adreßraum und hybride Speicherarchitekturen betrachtet. Der Schwerpunkt liegt dabei auf der Optimierung der Speicherzugriffslokalität, da moderne Prozessoren nur dann effizient genutzt werden können, wenn es gelingt, Wartezeiten aufgrund von Zugriffen auf den langsameren Speicher zu minimieren. Dies gilt insbesondere auf Parallelrechnern mit gemeinsamem Adreßraum, auf denen häufig das Speichersystem als begrenzte Hardware-Ressource die Skalierbarkeit speicherintensiver Programme limitiert.

Die Arbeit diskutiert zunächst verschiedene Möglichkeiten zur sequentiellen und parallelen Implementierung eingebetteter Runge-Kutta-Verfahren, die sich aufgrund unterschiedlicher Schleifenstrukturen insbesondere durch ihr Lokalitätsverhalten unterscheiden. Diese Implementierungsvarianten werden bezüglich ihres Laufzeit- und Lokalitätsverhaltens sowie im parallelen Fall auch ihres Skalierbarkeitsverhaltens analysiert und miteinander verglichen. Zu diesem Zweck werden sowohl theoretische Betrachtungen als auch Simulations- und Laufzeitexperimente herangezogen. Die Ergebnisse dieser Untersuchungen machen die Bedeutung von Lokalitätsoptimierungen für Laufzeit und Skalierbarkeit eingebetteter Runge-Kutta-Verfahren deutlich. Sie zeigen aber auch, daß die bisher betrachteten Programmtransformationen noch nicht zu einer zufriedenstellenden Skalierbarkeit bzw. Effizienz führen. Es werden deshalb zwei Ansätze vorgeschlagen, die in vielen Fällen zu einer Verbesserung der Laufzeit bzw. der Skalierbarkeit führen können.

Der erste Ansatz besteht darin, Wartezeiten, die bei einer parallelen Ausführung an Synchronisationspunkten auftreten, durch den Einsatz von Techniken zur dynamischen Lastbalancierung zu reduzieren. Um

die Anwendbarkeit dieses Ansatzes zu zeigen, werden Möglichkeiten zur Realisierung unterschiedlicher Lastbalancierungsstrategien diskutiert und verschiedene Implementierungen von Lastbalancierungsstrategien für Parallelrechner mit gemeinsamem Adreßraum in ausführlichen Laufzeitexperimenten untersucht. Die Ergebnisse der Experimente zeigen, daß mit Hilfe von Lastbalancierungstechniken in vielen Fällen sogar für ausgeglichene Testprobleme ein Laufzeitgewinn erzielt werden kann.

Der zweite Ansatz nutzt die spezifischen Datenabhängigkeiten der Funktionsauswertung von gewöhnlichen Differentialgleichungssystemen mit beschränkter Zugriffsdistanz, um weiterführende Programmtransformationen zu ermöglichen. Diese erlauben eine Abkehr von der sequentiellen Berechnung der Stufen und können so zu einer Verbesserung des Lokalitätsverhaltens sowie zu einer effizienteren Gestaltung der Kommunikation zwischen den beteiligten Kontrollflüssen ausgenutzt werden. Dadurch kann sowohl im sequentiellen als auch im parallelen Fall eine höhere Effizienz erreicht werden. Das vorgeschlagene Berechnungsschema hat weiterhin den Vorteil, daß eine Realisierung mit geringerem Speicherplatzbedarf gegenüber bisher vorgeschlagenen Verfahren möglich ist, ohne daß Einschränkungen bezüglich der Auswahl der Verfahrenskoeffizienten notwendig sind. Darüber hinaus erlaubt das vorgeschlagene Berechnungsschema eine Einsparung von Rechenzeit durch eine Modifikation der Schrittweitenkontrolle, die frühzeitig erkennt und eingreift, wenn ein Zeitschritt verworfen werden muß. Es wurden verschiedene Realisierungen des vorgeschlagenen Berechnungsschemas für unterschiedliche Zielarchitekturen erstellt und ausführlich mit Hilfe von theoretischen und simulationsbasierten Methoden sowie mit Hilfe von Laufzeitexperimenten analysiert. Die Ergebnisse bestätigen, daß das neue Berechnungsschema bereits auf sequentiellen Rechnersystemen durch ein verbessertes Lokalitätsverhalten zu einer deutlichen Reduzierung der Laufzeit führen kann. Auf parallelen Rechnersystem erlaubt das verbesserte Lokalitätsverhalten und der reduzierte Kommunikationsaufwand eine erhebliche Verbesserung der Skalierbarkeit. Das vorgeschlagene Berechnungsschema ist auf eine große Klasse gewöhnlicher Differentialgleichungssysteme anwendbar, u. a. viele Probleme, die durch eine Diskretisierung mittels Linienmethode aus partiellen Differentialgleichungssystemen hervorgehen, und es läßt sich z. B. auf iterierte Runge-Kutta-Verfahren übertragen.

1.5 Aufbau und Gliederung der Arbeit

Im aktuellen Kapitel wurde eine Motivation für die Durchführung dieser Arbeit gegeben, indem die betrachtete Problemklasse und mathematische Verfahren zu deren Lösung vorgestellt wurden und die Bedeutung der Speicherzugriffslokalität zur effizienten Nutzung moderner Rechnersysteme hervorgehoben wurde. Der Hauptteil dieser Arbeit gliedert sich wie folgt:

- Kapitel 2: Dieses Kapitel beschäftigt sich mit der sequentiellen Implementierung eingebetteter Runge-Kutta-Verfahren. Es wird zunächst darauf eingegangen, welche Entwurfsaspekte bei der Umsetzung des mathematischen Verfahrens in ein Programm zu berücksichtigen sind, und es werden allgemeine Ansätze zur Laufzeitverbesserung beschrieben. Darauf aufbauend werden Ansätze zur Implementierung eingebetteter Runge-Kutta-Verfahren unter dem Gesichtspunkt der Laufzeitminimierung diskutiert, es wird ein Rahmenprogramm entwickelt, und es werden verschiedene Implementierungsvarianten mit unterschiedlichem Lokalitätsverhalten vorgestellt. Diese werden mit Hilfe theoretischer Betrachtungen sowie mit Hilfe von Simulations- und Laufzeitexperimenten ausführlich analysiert und einander gegenübergestellt.

- Kapitel 3: Ausgehend von den anhand der Untersuchungen der sequentiellen Implementierungen gewonnenen Erkenntnissen betrachtet dieses Kapitel die Implementierung eingebetteter Runge-Kutta-Verfahren für Parallelrechner mit verteiltem und gemeinsamem Adreßraum sowie für Parallelrechner mit hybrider Speicherarchitektur. Es werden verschiedene Implementierungsvarianten vorgestellt und untersucht. Zur Verbesserung der Skalierbarkeit wird der Einsatz von Techniken zur dynamischen Lastbalancierung vorgeschlagen.

- Kapitel 4: In diesem Kapitel werden Möglichkeiten betrachtet, um durch eine Spezialisierung bezüglich des zu integrienden Problems weitere Laufzeitverbesserungen zu erzielen. Konkret werden solche Probleme betrachtet, bei denen die Funktionsauswertung einer Komponente der rechten Seite des gewöhnlichen Differentialgleichungssystems nur eine kleine Anzahl von Komponenten des Argumentvektors

innerhalb eines beschränkten Indexbereiches benötigt. Es wird ein Berechnungsschema vorgestellt, daß die Ableitung effizienter sequentieller und paralleler Implementierungen mit verbessertem Lokalitätsverhalten und reduziertem Kommunikationsaufwand ermöglicht. Verschiedene Implementierungsvarianten dieses Berechnungsschemas werden ausführlich untersucht und sowohl untereinander als auch mit in den vorhergehenden Kapiteln betrachteten allgemeinen Implementierungsvarianten verglichen. Weiterhin werden Möglichkeiten zur Reduzierung des Berechnungsaufwandes der Schrittweitenkontrolle sowie zur Einsetzbarkeit von Lastbalancierungstechniken diskutiert.

Kapitel 5 faßt abschließend die im Rahmen dieser Arbeit durchgeführten Untersuchungen und deren Ergebnisse zusammen und gibt einen Ausblick auf mögliche zukünftige Forschungsziele, die im Zusammenhang mit dieser Arbeit stehen. Eine ausführliche Beschreibung der verwendeten Testprobleme sowie der zur Durchführung von Laufzeitexperimenten genutzten Rechnersysteme ist als Anhang beigefügt.

2 Sequentielle Implementierung eingebetteter Runge-Kutta-Verfahren

In diesem Kapitel werden sequentielle Implementierungen eingebetteter Runge-Kutta-Verfahren für die Lösung von Anfangswertproblemen (1.1) besprochen. Ausgangspunkt für die Implementierung ist die Berechnungsvorschrift (1.9), ohne daß einschränkende Annahmen bezüglich der Eigenschaften des Differentialgleichungssystems oder der Verfahrenskoeffizienten getroffen werden. Bevor jedoch auf Implementierungsdetails eingegangen wird, werden zuvor einige grundsätzliche Überlegungen zur Umsetzung der mathematischen Verfahrensvorschrift in ein Computerprogramm dargelegt und allgemeine Ansätze besprochen, die eine Beschleunigung von Computerprogrammen ermöglichen können. Anschließend werden verschiedene Implementierungsmöglichkeiten für eingebettete Runge-Kutta-Verfahren vorgestellt und ihre Auswirkungen auf das Lokalitätsverhalten und die dadurch beeinflußte Effizienz der Programme diskutiert. Abschließend erfolgt ein Vergleich verschiedener Implementierungsvarianten anhand von Laufzeitexperimenten, die auf mehreren aktuellen Rechnersystemen durchgeführt wurden.

2.1 Vom mathematischen Verfahren zum Programm

Bevor man mit der Implementierung eines eingebetteten Runge-Kutta-Verfahrens beginnen kann, sind eine Reihe von Entwurfsentscheidungen zu treffen. Der folgende Abschnitt liefert einen Überblick über einige der wichtigsten Gesichtspunkte, die zu berücksichtigen sind.

2.1.1 Wahl der Programmiersprache

Eine der ersten Entwurfsentscheidungen ist die Wahl einer geeigneten Programmiersprache. Prinzipiell läßt sich die Verfahrensvorschrift der eingebetteten Runge-Kutta-Verfahren natürlich in nahezu jeder beliebigen Programmiersprache ausdrücken. Allerdings hat jede Programmiersprache ihre spezifischen Vor- und Nachteile, so daß die speziellen Entwurfsziele, die man mit der Implementierung verfolgt, bestimmte Typen von Programmiersprachen erfordern oder ausschließen können.

Da in dieser Arbeit Implementierungen mit möglichst geringer Laufzeit gesucht werden, wobei die Optimierung des Lokalitätsverhaltens der Implementierungen im Vordergrund steht, müssen die in dieser Arbeit betrachteten Implementierungen folgende Anforderungen erfüllen:

1. Sie müssen auf einer Vielzahl von Rechnersystemen lauffähig sein, um ein möglichst breites Spektrum unterschiedlicher Rechnerarchitekturen für Laufzeitexperimente nutzen zu können.

2. Um eine möglichst hohe Ausführungsgeschwindigkeit zu erreichen, sollen die Programme in die Maschinensprache des jeweiligen Rechnersystems übersetzt werden können.

3. Um dem Programmierer die weitestmögliche Kontrolle über den Programmfluß des Maschinenprogramms, insbesondere über die Reihenfolge der ausgeführten Speicherzugriffsoperationen zu ermöglichen, sollen die Quellprogramme die Maschinensprache nur gerade soweit abstrahieren, daß die unter Punkt 1 geforderte Portabilität gewährleistet ist.

Um diese Forderungen erfüllen zu können, ist eine systemnahe, imperative Programmiersprache erforderlich, für die effiziente, hoch-optimierende Compiler verfügbar sind. Um auch die in den nachfolgenden

Kapiteln betrachteten parallelen Implementierungen auf einer möglichst großen Anzahl paralleler Rechnersysteme nutzen zu können, sollte die eingesetzte Programmiersprache die Nutzung der gebräuchlichsten Anwendungsschnittstellen für den Datenaustausch zwischen parallel ausgeführten Kontrollflüssen erlauben.

Zwei Programmiersprachen, die all dies leisten können, sind FORTRAN und C. Beide Programmiersprachen sind imperative Sprachen, deren Quellprogramme eine Folge von Anweisungen enthalten, die durch den Compiler in eine äquivalente Folge von Maschinenbefehlen transformiert werden. Für beide Sprachen sind hoch-optimierende Compiler für nahezu alle gebräuchlichen Plattformen verfügbar. Im Gegensatz zu anderen imperativen Sprachen bieten FORTRAN und C den Vorteil, daß man im Unix-Umfeld eine besonders gute Unterstützung für diese Sprachen vorfindet. Da aktuelle Installationen von Supercomputern in der Regel ein Unix-Derivat als Betriebssystem verwenden, kann man somit davon ausgehen, auf nahezu allen derartigen Systemen geeignete Compiler vorzufinden. Oft werden vom Hersteller dieser Systeme sogar systemspezifische Compiler für C und FORTRAN mitgeliefert, die erfahrungsgemäß durch zielgerichtetere Optimierungen effizienteren Code erzeugen können als andere Compiler, die eine Vielzahl von Plattformen unterstützen. Standardisierte Anwendungsschnittstellen für den Datenaustausch zwischen parallel ausgeführten Kontrollflüssen, wie z. B. MPI (MPI Forum 1995, 1997), OpenMP (OpenMP API 2.5 2005) und POSIX Threads (Butenhof 1997), existieren für beide Sprache.

FORTRAN wurde im Jahr 1954 entwickelt und ist eine der ältesten Programmiersprachen. Wie es der Name „FORTRAN“ – „FORmula TRANsactor“ bereits andeutet, wurde FORTRAN für numerische Berechnungen konzipiert. So gibt es in FORTRAN z. B. einen Potenzoperator und einen Datentyp für komplexe Zahlen. FORTRAN ist u. a. deshalb bis heute bei Entwicklern numerischer Software sehr beliebt. Die Programmiersprache C wurde dagegen erst Anfang der 70er Jahre für das neu entwickelte Betriebssystem Unix entworfen. Die Unix-Kernel sowie die Kernel vieler anderer Betriebssysteme wurden in C implementiert. C eignet sich deshalb sehr gut für systemnahe Programmierung, weil es einen einfachen Zugang zu Systemfunktionen des Kernels ermöglicht. Für die Lokalitätsoptimierung von Programmen ist von besonderer Bedeutung, daß C durch die Bereitstellung eines Zeiger-Datentyps sowie durch die Möglichkeit der dynamischen Speicherallokierung eine gezieltere Einflußnahme auf die Anordnung der Datenstrukturen im Speicher sowie das resultierende Speicherzugriffsmuster des Programms ermöglicht. In dieser Arbeit wird daher die Programmiersprache C verwendet, weil sie die oben genannten Forderungen am besten erfüllt.

Gelten andere Anforderungen an die zu erstellenden Implementierungen, kann sich die Wahl einer anderen Programmiersprache anbieten. Ist etwa bestmögliche Kompatibilität zu existierenden FORTRAN-Codes gefordert, so ist natürlich FORTRAN die Sprache der Wahl. Für die Realisierung komplexer Softwaresysteme kann eine objektorientierte Sprache, wie z. B. C++, am geeignetsten sein. Eine alternative objektorientierte Sprache, die darüber hinaus ein zusätzliches Maß an Plattformunabhängigkeit bietet, ist z. B. Java. Experimente mit parallelen Implementierungen iterierter Runge-Kutta-Verfahren haben beispielsweise gezeigt, daß Java-Implementierungen eine ähnlich gute Skalierbarkeit erreichen können wie vergleichbare Implementierungen in C, die POSIX Threads zur Synchronisation nutzen (Korch u. Rauber 2004a).

2.1.2 Entwurfsaspekte für die Realisierung des numerischen Verfahrens

Bezüglich der Auswahl und der Umsetzung des numerischen Verfahrens stellen sich eine Reihe von Fragen, die einer Antwort bedürfen, bevor man mit der Realisierung eines Programms beginnen kann. Dazu gehört eine genau Festlegung, welche Eingabedaten vorliegen und welche Ausgaben das Programm berechnen soll, aber auch, wie die Anwendungsschnittstelle der Software aufgebaut sein soll, welche Struktur der Programmcode besitzen soll und welche Datenstrukturen erforderlich sind.

Programmtechnische Realisierung einer gewöhnlichen Differentialgleichung

Eine der ersten zu beantwortenden Fragen ist, wie man ein gewöhnliches Differentialgleichungssystem in Programmform realisiert. Nach (1.1) ist ein nichtautonomes Anfangswertproblem einer gewöhnlichen Differentialgleichung gegeben durch

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)) \ , \qquad \mathbf{y}(t_0) = \mathbf{y}_0 \ .$$

Das Differentialgleichungssystem wird dabei beschrieben durch die Funktion $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$, die eine reelle Zahl sowie einen n-dimensionalen reellwertigen Vektor als Argumente besitzt und daraus einen n-dimensionalen reellwertigen Vektor als Ergebnis der Funktionsauswertung berechnet. Es liegt nahe, die mathematische Funktion $\mathbf{f}$ durch eine Funktion im Sinne der verwendeten Programmiersprache zu realisieren, die mit entsprechenden Parametern und Rückgabewerten ausgestattet wird. Einige Dinge sind hierbei jedoch zu beachten.

Unterstützung mehrerer Differentialgleichungen. Soll es möglich sein, mehrere verschiedene Differentialgleichungen mit rechten Seiten $\mathbf{f}_1, \mathbf{f}_2, \ldots$ mit Hilfe des implementierten Lösungsverfahrens zu lösen, gibt es verschiedene Möglichkeiten, dies programmtechnisch umzusetzen.

1. *Verwendung des gleichen Bezeichners* für die Namen aller Funktionen $\mathbf{f}_1, \mathbf{f}_2, \ldots$ innerhalb des Programms. Da es in einigen Programmiersprachen (u. a. C und FORTRAN) nicht erlaubt ist, daß innerhalb eines Programms mehrere Funktionen mit gleichem Bezeichner vorhanden sind,[1] müssen die Funktionen für die verschiedenen Differentialgleichungen in separaten Quelldateien abgelegt werden. Diese Quelldateien können einzeln in Objektcode übersetzt, jedoch nicht statisch zu einem gemeinsamen Binärprogramm gebunden werden. Dieser Ansatz wird in vielen klassischen ODE-Codes verwendet (siehe auch Abschnitt 2.1.3).

   Möchte man nur ein einziges Binärprogramm bereitstellen, dann ist deshalb zum Wechseln der Differentialgleichung ein erneutes Binden des Binärprogramms erforderlich. Alternativ könnte man für jede Differentialgleichung ein separates Binärprogramm erzeugen. Eine weitere Möglichkeit wäre, die rechten Seiten der Differentialgleichungen als separate dynamische Bibliotheken zu implementieren und die jeweilige Differentialgleichung durch dynamisches Binden der entsprechenden Bibliothek auszuwählen.

   Der Vorteil der Verwendung nur eines Bezeichners besteht darin, daß innerhalb des Programmteils, der das Lösungsverfahren realisiert, dieser Bezeichner direkt zum Aufruf der entsprechenden Programmfunktion verwendet werden kann, ohne daß neben den Kosten für den Funktionsaufruf (Sprung zur Adresse der Funktion, evtl. Sichern und Wiederherstellen von Registern, Rücksprung) weitere Kosten entstehen.

2. *Verwendung verschiedener Bezeichner* für die Funktionsnamen, z. B. durch Verwendung eines geeigneten Präfixes wie etwa dem Namen der Differentialgleichung. In diesem Fall ist es problemlos möglich, alle Differentialgleichungen innerhalb eines einzelnen Binärprogramms zur Verfügung zu stellen. Allerdings muß man jetzt dafür Sorge tragen, daß innerhalb des Lösungsalgorithmus die passende Funktion aufgerufen wird.

   Dazu benötigt man eine zusätzliche Variable, die die Information enthält, welches Differentialgleichungssystem gelöst werden soll. Mit Hilfe dieser Information könnte man z. B. jede Funktionsauswertung mit einer Fallunterscheidung verbinden, die allerdings mit unnötig hohen Kosten (mehrfaches Auslesen und Vergleichen der Variablen) verbunden ist. Hinzu kommt, daß, wenn eine zusätzliche Differentialgleichung implementiert werden soll, alle Programmstellen, an denen eine solche Funktionsauswertung aufgerufen wird, modifiziert werden müssen.

   Um die mit der Fallunterscheidung verbundenen Probleme zu umgehen, kann man die zusätzliche Variable dazu nutzen, die Adresse der Programmfunktion zu speichern, welche die rechte Seite der zu lösenden Differentialgleichung realisiert. Dann ist es möglich, die ausgewählte Funktion über diesen Funktionszeiger indirekt aufzurufen. Im Vergleich zur Verwendung gleicher Bezeichner entstehen hier zwar ebenfalls zusätzliche Kosten, da ein Speicherzugriff und ein indirekter Sprung erforderlich sind, um den Inhalt der Variablen zu ermitteln und die vorgefundene Adresse anzuspringen, diese sind aber wesentlich geringer als bei Verwendung einer Fallunterscheidung.

Bereitstellung von Komponentenfunktionen. Bei der Realisierung der rechten Seite als Programmfunktion hat man die Möglichkeit, die Programmfunktion entweder als Vektorfunktion $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ zu gestalten, so daß ein einzelner Funktionsaufruf alle Komponenten des Ergebnisvektors berechnet, oder sie

[1] In einigen anderen Programmiersprachen ist dies erlaubt, wenn die Funktionen anhand der Parameterliste unterscheidbar sind. Dies ist hier allerdings nicht der Fall, da die Funktionen $\mathbf{f}_1, \mathbf{f}_2, \ldots$ die gleichen Argumente und Rückgabewerte besitzen.

als skalare Funktion $f : \mathbb{N} \times \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}$ zu implementieren, die jeweils nur eine einzelne Vektorkomponente berechnet.

Wenn die Berechnungsvorschrift für die rechte Seite des Differentialgleichungssystems dies zuläßt, bietet die Bereitstellung von skalaren Komponentenfunktionen eine Reihe von Vorteilen gegenüber der Implementierung als Vektorfunktion. So ermöglicht die Aufsplittung in Komponentenfunktionen u. a. eine datenparallele Auswertung der rechten Seite, indem auf jedem der verwendeten Prozessoren jeweils eine Teilmenge der Komponentenfunktionen berechnet wird. Beim Entwurf aller Implementierungen – sequentiell oder parallel – erlaubt die Verwendung von Komponentenfunktionen die Durchführung von Programmtransformationen, die andernfalls nicht möglich wären. Die Verwendung von Komponentenfunktionen ist daher die Voraussetzung für einen Großteil der Programmtransformationen zur Optimierung des Speicherzugriffsverhaltens vieler sequentieller und paralleler Implementierungen, die in dieser Arbeit betrachtet werden.

Programmtechnische Realisierung des Integrationsalgorithmus

Die Realisierung des Integrationsalgorithmus setzt eine Festlegung der gewünschten Ausgaben und der dafür erforderlichen Eingangsinformationen voraus. Darüber hinaus ist zu überlegen, welches konkrete numerische Verfahren geeignet ist, um die an die Qualität der Lösung gestellten Anforderungen zu erfüllen, und welches gleichzeitig hinreichend effizient arbeitet, um die gesuchte Lösung innerhalb einer angemessenen Antwortzeit zu liefern.

Nach (1.1) werden Probleme aus der betrachteten Klasse nichtautonomer Anfangswertprobleme beschrieben durch

$$t \in [t_0, t_e] : \qquad \mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)) \ , \qquad \mathbf{y}(t_0) = \mathbf{y}_0 \ .$$

Gegeben ist also ein Intervall $[t_0, t_e]$, eine Funktion $\mathbf{f}$, deren programmtechnische Realisierung bereits angesprochen wurde, sowie der Anfangswert $\mathbf{y}_0$. Gesucht wird eine Approximation der Lösungsfunktion $\mathbf{y}(t)$, deren Ableitung im Intervall $[t_0, t_e]$ durch die Funktion $\mathbf{f}$ beschrieben wird. Es kann auch vorkommen, daß der Endpunkt des Intervalls, t_e, zu Beginn der Integration nicht bekannt ist, da nach einer Stelle gesucht wird, an der ein bestimmtes Ereignis eintritt, wie z. B. das Erfülltsein einer bestimmten Bedingung für t und $\mathbf{y}(t)$.

Die Berechnung der Approximation der Lösungsfunktion erfolgt in der Regel in Form einer Gitterfunktion

$$\eta_\kappa = \eta(t_\kappa) \approx \mathbf{y}(t_\kappa) \ , \qquad \kappa = 0, \ldots, N \ ,$$

die Näherungswerte für Funktionswerte von $\mathbf{y}(t)$ an mehreren Zwischenstellen

$$t_0 < t_1 < \cdots < t_N = t_e \tag{2.1}$$

des Intervalls $[t_0, t_e]$ liefert (vgl. (1.4) und (1.5)). Anzahl und Position der Zwischenstellen werden von der Schrittweitenkontrolle bestimmt. Die Ausgabe der berechneten Lösung kann je nach Anwendungsfall auf sehr unterschiedliche Weise erfolgen. Oft genügt die Ausgabe der Approximationswerte an den Gitterstellen $t_0, \ldots, t_e$. In bestimmten Fällen kann sogar nur der einzelne Approximationswert für $\mathbf{y}(t_e)$ am Rand des Intervalls gesucht sein. Einige Anwendungen können jedoch auch die Ausgabe von Approximationswerten an wesentlich dichter liegenden Zwischenstellen erfordern, als sie von der Schrittweitenkontrolle generiert werden, z. B. für eine grafische Darstellung der Lösungsfunktion.

Nahezu jede Implementierung eines Lösungsverfahrens mit Schrittweitenkontrolle gibt dem Anwender die Möglichkeit, durch Vorgabe eines oder mehrerer Parameter, die oft mit TOL bzw. ATOL und RTOL bezeichnet werden, Einfluß darauf zu nehmen, wie groß der geschätzte Betrag des lokalen Fehlers sein darf, damit ein Zeitschritt von der Schrittweitenkontrolle akzeptiert wird (vgl. Abschnitt 1.1.1). Dadurch ist es für den Anwender möglich, den für seinen speziellen Anwendungsfall besten Kompromiß zwischen Genauigkeit und Laufzeit auszuwählen.

Neben der Ausgabe der Approximation der Lösungsfunktion können für den Anwender eine Reihe zusätzlicher Informationen über den Integrationsvorgang von Interesse sein, etwa über die Größe des lokalen und globalen Fehlers und über die durchgeführten Zeitschritte. Kann die Integration nicht erfolgreich abgeschlossen werden, weil z. B. aus numerischen Gründen die Integration nicht fortgesetzt werden kann oder weil eine Ausnahmebehandlung erforderlich ist, sollte der Anwender detaillierte Informationen über die Ursache erhalten.

Wahl der Verfahrenskoeffizienten. Der Kern jeder Implementierung eines eingebetteten Runge-Kutta-Verfahrens besteht in der Umsetzung der mathematischen Berechnungsvorschrift des Verfahrens (vgl. (1.9), (1.16), (1.17)):

$$\mathbf{v}_l = \mathbf{f}\left(t_\kappa + c_l h_\kappa, \eta_\kappa + h_\kappa \sum_{i=1}^{l-1} a_{li}\mathbf{v}_i\right) \quad \text{für} \quad l = 1, \ldots, s \ , \tag{2.2a}$$

$$\eta_{\kappa+1} = \eta_\kappa + h_\kappa \sum_{l=1}^{s} b_l \mathbf{v}_l \ , \tag{2.2b}$$

$$\hat{\eta}_{\kappa+1} = \hat{\eta}_\kappa + h_\kappa \sum_{l=1}^{s} \hat{b}_l \mathbf{v}_l \ . \tag{2.2c}$$

Die Koeffizienten a_{li}, b_l, $\hat{b}_l$ und c_l definieren das konkrete Verfahren und bestimmen dessen numerische Eigenschaften. Sie sollten deshalb sorgfältig anhand der Eigenschaften der zu lösenden Differentialgleichungssysteme und der Anforderungen an die zu berechnende Approximation der Lösungsfunktion ausgewählt werden. Verschiedene eingebettete Verfahren werden z. B. in Hairer u. a. (2000) und Butcher (2003) bezüglich ihrer numerischen Eigenschaften diskutiert und gegenübergestellt.

Die Entscheidung, welche Verfahrenskoeffizienten unterstützt werden sollen, hat weiterhin einen entscheidenden Einfluß auf die Möglichkeiten zur programmtechnischen Realisierung des Verfahrens in Form einer Implementierung. Soll die Implementierung beliebige eingebettete Verfahren unterstützen, müssen die Verfahrenskoeffizienten in dynamischen Datenstrukturen (z. B. ein- bzw. zweidimensionale Felder) abgelegt werden, deren Größe und Inhalt erst nach Vorgabe durch den Benutzer zur Laufzeit bekannt ist. Die Berechnungsvorschrift (2.2) muß in diesem Fall mit Hilfe von Schleifen realisiert werden, deren Grenzen von der Stufenzahl s und der Größe des Differentialgleichungssystems n abhängen und ebenfalls erst zur Laufzeit bekannt sind.

Zusätzliche Möglichkeiten zur Optimierung des Programmcodes kann man durch eine Spezialisierung erreichen, indem nur bestimmte Verfahrensklassen oder sogar nur ein einziges Verfahren zugelassen werden. Legt man z. B. allein die Anzahl der Stufen fest, sind die Anzahl der Verfahrenskoeffizienten und die Grenzen der Schleifen, die über die Stufen iterieren, bereits zur Übersetzungszeit bestimmt und können für Optimierungen ausgenutzt werden. So kann man etwa die Verfahrenskoeffizienten in statischen Datenstrukturen (z. B. statische Felder oder skalare Variablen) vorhalten und durch Aufrollen von Schleifen zusätzliche Codetransformationen ermöglichen.

Einige eingebettete Runge-Kutta-Verfahren besitzen die sogenannte FSAL-Eigenschaft (englisch: *first same as last*). Das bedeutet, daß die Koeffizienten $b_1, \ldots, b_{s-1}$ identisch mit den Koeffizienten $a_{s,1}, \ldots, a_{s,s-1}$ sind und $b_s = 0$ gilt. Infolgedessen stimmen auch der Vektor $\mathbf{v}_s$ aus dem Zeitschritt κ und der Vektor $\mathbf{v}_1$ aus dem nachfolgenden Zeitschritt $\kappa + 1$ überein, so daß eine Auswertung der Funktion $\mathbf{f}$ eingespart werden kann. Bei einer Spezialisierung der Implementierung auf Verfahren mit der FSAL-Eigenschaft ist demzufolge ein signifikanter Laufzeitgewinn im Vergleich zur Ausführung derselben Verfahren mittels einer allgemeineren Implementierung zu erwarten.

Legt man sich beim Entwurf der Implementierung auf nur ein spezielles Verfahren fest, ergeben sich die umfangreichsten Möglichkeiten zur Optimierung. Da die Beträge der Verfahrenskoeffizienten bekannt sind, können diese nun direkt als Konstanten in den Quellcode eingefügt werden. Haben einige Koeffizienten den Betrag 0 oder 1, können Rechenoperationen eingespart werden, indem Additionen mit 0 und Multiplikationen mit 0 oder 1 vermieden werden. Dies kann dadurch erfolgen, daß man diese Operationen erst gar nicht in den Programmcode aufnimmt oder daß ein optimierender Compiler diese Spezialfälle erkennt und effizienten Maschinencode generiert.

Vermeidung von Rundungsfehlern. Bei der Implementierung jedes numerischen Algorithmus sollte man versuchen, Rundungsfehler so weit als möglich zu minimieren. Die Verfahrensvorschrift für eingebettete Runge-Kutta-Verfahren (2.2) enthält neben den Funktionsauswertungen der rechten Seite des Differentialgleichungssystems ausschließlich Additions- und Multiplikationsoperationen, die zur Entstehung des Rundungsfehlers beitragen. Besonders kritisch ist die Berechnung von $\eta_{\kappa+1}$ nach (2.2b) und $\hat{\eta}_{\kappa+1}$ nach (2.2c), da die Komponenten der Vektoren $h_\kappa \sum_{l=1}^{s} b_l \mathbf{v}_l$ bzw. $h_\kappa \sum_{l=1}^{s} \hat{b}_l \mathbf{v}_l$ in der Regel sehr klein gegenüber den

Komponenten $\eta_{\kappa+1}$ bzw. $\hat{\eta}_{\kappa+1}$ sind. Die schlimmsten Auswirkungen der Akkumulation von Rundungsfehlern können jedoch vermieden werden, indem man den bei dieser Addition auftretenden Fehler abschätzt und ihn im nächsten Zeitschritt zum kleineren Summanden hinzuaddiert (Butcher 2003). Diese Technik bezeichnet man als *kompensierte Summation* (englisch: *compensated summation*). Higham (1993) gibt einen ausführlichen Überblick über verschiedene Ansätze für eine möglichst genaue Summation von Gleitkommazahlen.

Stetige Erweiterungen. Einige Anwendungen benötigen Näherungslösungen für die Werte der Lösungsfunktion außerhalb des von der Schrittweitenkontrolle erzeugten Punktgitters. Beispielsweise könnte für die Erzeugung einer grafischen Darstellung der Lösungsfunktion ein festes Punktgitter vorgegeben sein, dessen Gitterpunkte sehr dicht liegen können. Um die Lösung an allen geforderten Zwischenpunkten zu berechnen, könnte man die Schrittweite entsprechend reduzieren. Da hierdurch gleichzeitig die Anzahl der Zeitschritte und damit verbunden die Anzahl der Funktionsauswertungen ansteigt, ist dieses Vorgehen allerdings sehr ineffizient. Weiterhin führt die Erhöhung der Anzahl der Zeitschritte zu einer Zunahme von Rundungsfehlern.

Es wurden deshalb Runge-Kutta-Verfahren konstruiert, die für jeden Zeitschritt κ Approximationen für alle Werte $\mathbf{y}(t_\kappa + \theta h_\kappa)$ im Bereich $0 \leq \theta \leq 1$ liefern, wobei dafür keine oder nur wenige zusätzliche Funktionsauswertungen erforderlich sind. Man bezeichnet derartige Verfahren als *stetige Runge-Kutta-Verfahren*. Zum Teil wird in diesem Zusammenhang auch der Begriff *stetige Erweiterung* benutzt, da eine Reihe stetiger Verfahren aus klassischen Verfahren durch das Hinzufügen zusätzlicher Stufen konstruiert wurden. Die Grundlage für die Berechnung der Zwischenwerte bilden spezielle Interpolationsansätze. Eine ausführliche Darstellung sowie weiterführende Literaturhinweise geben z. B. Hairer u. a. (2000) sowie Enright u. a. (1995).

Anfangsschrittweite. Eine Möglichkeit zur Festlegung der Anfangsschrittweite besteht darin, vom Anwender eine Vorgabe der Anfangsschrittweite zu fordern. Dies setzt voraus, daß der Anwender aus seiner Erfahrung oder aus mathematischen Überlegungen heraus eine Vorstellung von der zu wählenden Größe der Anfangsschrittweite besitzt. Um eine ungünstige Wahl der Anfangsschrittweite durch den Anwender zu vermeiden, ist eine bessere Vorgehensweise, die Anfangsschrittweite durch einen geeigneten Algorithmus festzulegen bzw. dem Anwender einen entsprechenden Vorschlag zu unterbreiten. Ein möglicher Algorithmus zur Bestimmung der Anfangsschrittweite wurde von Gladwell u. a. (1987) vorgeschlagen (siehe auch Hairer u. a. 2000).

Behandlung des Intervallendes. Zur Erzeugung des Punktgitters (2.1) wird die Schrittweite in jedem Zeitschritt unter Beachtung der Genauigkeitsforderung möglichst groß gewählt, um die Anzahl der Zeitschritte und damit den Rechenaufwand zu minimieren. Wurden die Gitterpunkte $t_0, \ldots, t_{N-1}$ auf diese Weise erzeugt, ergibt sich die Schrittweite für den letzten Zeitschritt zur Bestimmung des Ausgabepunktes am Intervallende $t_e = t_N$ als $h_{N-1} = t_N - t_{N-1}$. Die Schrittweite wird in diesem Zeitschritt also kleiner gewählt, als es aufgrund der Genauigkeitsforderung notwendig wäre, um den Ausgabepunkt am Intervallende ermitteln zu können. Dies kann zu einem erhöhten Rechenaufwand führen, insbesondere wenn die Integration auf dem sich an t_e anschließenden Intervall fortgesetzt wird und dieses Vorgehen systematisch für weitere Intervalle fortgeführt wird.

Um die durch die Verkleinerung der letzten Schrittweite hervorgerufenen Effekte abzumildern, kann man versuchen, das Intervallende „vorauszuahnen“ und die Schrittweite bereits in früheren Zeitschritten zu reduzieren. Ein anderer Ansatz besteht in der Verwendung eines stetigen Runge-Kutta-Verfahrens. In diesem Fall wird die Schrittweite nicht eingeschränkt, und die Integration wird fortgesetzt, bis das Intervallende t_e erstmalig überschritten wird. Die Approximation des Funktionswerts $\mathbf{y}(t_e)$ wird dann durch Interpolation ermittelt (siehe Enright u. a. 1995).

Abbruchkriterien und besondere Ereignisse. In bestimmten Situationen ist es sinnvoll, die Integration bereits vor dem Erreichen des Intervallendes zu beenden bzw. neu zu starten. Eine solche Situation liegt z. B. vor, wenn erkannt wird, daß der Rechenaufwand zur Berechnung der gesuchten Lösung zu groß wäre, um die Berechnung innerhalb einer akzeptablen Zeitspanne zu beenden, oder wenn das Anwachsen von Rundungsfehlern zu befürchten ist. Mögliche Abbruchkriterien können sein:

1. Eine zu kleine Schrittweite, da diese zu einer großen Anzahl von Zeitschritten und damit zu einem hohen Berechnungsaufwand und der Akkumulation von Rundungsfehlern führt.

2. Die Berechnung einer vorgegebenen maximalen Anzahl von Zeitschritten.

3. Die Detektion von Steifheit (siehe Hairer u. Wanner 2002), da explizite Verfahren für die Integration steifer Differentialgleichungssysteme ungeeignet sind.

4. Das Auftreten einer Singularität (siehe Enright u. a. 1995).

5. Das Erfülltsein einer vom Nutzer vorgegebenen Bedingung (siehe Enright u. a. 1995; Hairer u. a. 2000; Shampine u. Thompson 2000).

6. Das Auftreten von Programmausnahmen, verursacht z. B. durch das Fehlschlagen von Betriebssystemaufrufen (etwa Anforderung von Speicher oder Ein-/Ausgabe-Funktionen).

Andere Situationen können eine spezielle Behandlung, z. B. einen Neustart oder einen Strategiewechsel der Schrittweitenkontrolle, erfordern. Dies ist beispielsweise der Fall, wenn innerhalb des Integrationsintervalls Unstetigkeiten auftreten oder sehr viele aufeinanderfolgende Schritte verworfen werden (siehe auch Enright u. a. 1995).

Statistische Informationen. Zusätzlich zur berechneten Näherungslösung des gestellten Anfangswertproblems können für den Anwender statistische Informationen über die Eigenschaften der Lösung sowie den Integrationsvorgang von Interesse sein. Beispielsweise könnte sich der Anwender für Aussagen über die Genauigkeit der berechneten Näherungslösung interessieren, also etwa den lokalen und globalen Fehler. Eine Approximation des lokalen Fehlers wird in jedem Zeitschritt berechnet, da sie für die Schrittweitenkontrolle benötigt wird. Zur Bestimmung des globalen Fehlers wurden verschiedene Techniken vorgeschlagen, siehe Enright u. a. (1995) für einen Überblick.

Um die Effizienz des Integrationsvorgangs einschätzen zu können, sind Informationen über die ausgeführten Zeitschritte, d. h. die Gesamtanzahl sowie der Anteil der akzeptierten bzw. verworfenen Schritte und die dazugehörigen Schrittweiten, hilfreich.

Für eine detaillierte Untersuchung des Laufzeit- und Lokalitätsverhaltens ist außerdem die Ausgabe diesbezüglicher Meßdaten erforderlich. Dabei kann es sich beispielsweise um Laufzeiten für verschiedene Programmteile oder Informationen über das Auftreten bestimmter Hardware-Ereignisse, wie z. B. Speicherzugriffe auf die verschiedenen Ebenen der Speicherhierarchie oder die Ausführung bestimmter Instruktionstypen, handeln.

Schnittstelle zum Anwendungsprogramm. Bei Verwendung einer nicht-objektorientierten, imperativen Programmiersprache wie C oder FORTRAN wird man ein Lösungsverfahren in der Regel in Form eines oder mehrerer Unterprogramme realisieren, um den Code des Lösungsverfahrens vom Anwendungsprogramm zu trennen und so den Code auch für andere Anwendungsprogramme nutzbar zu machen.

Für die Kommunikation zwischen dem Unterprogramm des Integrators und dem aufrufenden Anwendungsprogramm gibt es verschiedene Möglichkeiten:

- **Parameterübergabe:** Daten werden beim Aufruf bzw. der Rückkehr des Integrators als Parameter übergeben.

- **Globale Variablen:** Daten werden vom Anwendungsprogramm vor Aufruf des Integrators in globalen Variablen abgelegt, die später vom Integrator ausgelesen werden. Der Integrator legt Rückgabedaten in globalen Variablen ab, die nach Beendigung dem Anwendungsprogramm zur Verfügung stehen. Im einfachsten Fall kann dem Anwendungsprogramm der direkte Zugriff auf diese globalen Variablen gestattet werden. Sicherer ist es aber, dem Anwendungsprogramm das Setzen und Auslesen der Variablen nur über spezielle Funktionen zu gestatten, da so eine Überprüfung der Zugriffe möglich ist.

Soll während der Integration Programmcode des Anwendungsprogramms ausgeführt werden, kann eine der folgende Techniken eingesetzt werden:

- **Call-Back-Funktionen:** Das Anwendungsprogramm stellt Unterprogramme bereit, die vom Integrator aufgerufen werden. Daten können über Parameter bzw. Rückgabewerte ausgetauscht werden, aber auch der Datenaustausch über globale Variablen ist möglich.

- **Rücksprung zum Anwendungsprogramm:** Das Unterprogramm des Integrators wird zwischenzeitlich beendet, und das Anwendungsprogramm wird fortgesetzt. Der Status des Integrators wird entweder in globalen Variablen festgehalten oder als Rückgabe an das Anwendungsprogramm übergeben. Zur Fortsetzung der Integration ruft das Anwendungsprogramm den Integrator erneut auf.

- **Nebenläufige Ausführung:** Integrator und Anwendungsprogramm werden als separate Kontrollflüsse realisiert, die auf geeignete Weise miteinander kommunizieren.

Die Ausführung von Programmcode des Anwendungsprogramms während der Integration wird in der Praxis häufig eingesetzt, um nacheinander anfallende Zwischenergebnisse, wie z. B. die Approximationswerte η_κ, anwendungsspezifisch zu verarbeiten. Existierende Codes lassen sich meist diesbezüglich klassifizieren als:

- **Schrittorientierte Codes:** Der Integrator berechnet jeweils einen Zeitschritt und kehrt anschließend zum Anwendungsprogramm zurück. Zur Berechnung weiterer Zeitschritte muß der Integrator wiederholt aufgerufen werden.

- **Intervallorientierte Codes:** Es wird ein Intervall vorgegeben. Der Integrator berechnet so viele Zeitschritte, wie zum Durchlaufen des Intervalls erforderlich sind. Zur Verarbeitung von Zwischenergebnissen durch das Anwendungsprogramm werden Call-Back-Funktionen verwendet.

Shampine u. a. (1976) geben einen Überblick über die wichtigsten damals verbreiteten Codes für die Lösung nichtsteifer Anfangswertprobleme. Diese Autoren haben die Erfahrung gemacht, daß Anwender oft intervallorientierte Codes bevorzugen.

Nebenläufige Codes sind bisher kaum verbreitet, obwohl sie ein signifikantes Potential zur Laufzeitverbesserung bieten. Beispielsweise führt ein typisches Anwendungsprogramm eine Reihe von Ausgabeoperationen aus, z. B. zur grafischen Darstellung der Lösungsfunktion, zur textuellen Ausgabe auf dem Bildschirm oder zur Speicherung der Approximationswerte in einer Datei für die spätere Verarbeitung. Die Wartezeiten, die beim Zugriff auf die Ausgabegeräte entstehen, können bei einer nebenläufigen Realisierung verdeckt werden. Ein denkbares Vorgehen wäre eine Implementierung auf Basis zweier Threads, wobei der Integrationsthread die schrittweise berechneten Approximationswerte über einen Produzenten-Konsumenten-Mechanismus an den Anwendungsthread übermittelt. Eine derartige Realisierung wäre insbesondere für moderne Prozessoren, die ein hardwarebasiertes (simultanes) Multithreading oder sogar mehrere Ausführungskerne bieten, vielversprechend, da hier zusätzlich die parallele Nutzung verschiedener Funktionseinheiten möglich wäre.

2.1.3 Existierende Softwareimplementierungen eingebetteter RK-Verfahren

Im folgenden werden einige der bekanntesten sequentiellen Softwareimplementierungen eingebetteter Runge-Kutta-Verfahren vorgestellt. Für einen detaillierten Überblick sowie einen Vergleich dieser Implementierungen siehe z. B. Hairer u. a. (2000), Strehmel u. Weiner (1995), Enright u. a. (1995) und Shampine u. Gladwell (1996).

DOPRI5 und DOP853. Die Codes DOPRI5 und DOP853 wurden von E. Hairer und G. Wanner entwickelt (siehe Hairer u. a. 1987, 2000). DOPRI5 realisiert ein eingebettetes Verfahren der Ordnung 5(4), basierend auf Verfahrenskoeffizienten von Dormand u. Prince (1980). Die in DOP853 verwendeten Koeffizienten basieren auf Verfahrenskoeffizienten, die von Dormand u. Prince (1989) angekündigt wurden. Aufbauend darauf konstruierten Hairer u. a. (2000) ein Verfahren der Ordnung 8 mit zwei eingebetteten Lösungen der Ordnung 5 und 3. Die Codes arbeiten intervallorientiert und verfügen über eine stetige Erweiterung sowie eine Steifheitserkennung. Sie wurden in FORTRAN implementiert und später auch nach C und C++ portiert.

DVERK. Der Code DVERK von Hull u. a. (1976) implementiert ein eingebettetes Verfahren der Ordnung 6(5), vorgeschlagen von Verner (1978). Er wurde in die numerische Bibliothek IMSL (siehe IMSL-Homepage) integriert, die für die Programmiersprachen C, C#, Java und FORTRAN verfügbar ist.

RKSUITE. Die Sammlung RKSUITE wurde von Brankin u. a. (1992, 1993) entwickelt und in die Numerikbibliothek der Numerical Algorithms Group (NAG) integriert (siehe NAG-Webseite Numerikkomponenten). Sie enthält drei verschiedene eingebettete Verfahren mit der Ordnung 3(2) nach Bogacki u. Shampine (1989a), der Ordnung 5(4) nach Bogacki u. Shampine (1989b) und der Ordnung 8(7) nach Prince u. Dormand (1981). RKSUITE bietet eine sehr große Funktionalität, unter anderem stetige Erweiterungen für die beiden Verfahren der Ordnung 3(2) und 5(4) sowie eine Abschätzung des globalen Fehlers durch eine zusätzliche Integration mit höherer Genauigkeit.

Es werden zwei verschiedene Anwendungsschnittstellen angeboten: Eine einfach zu benutzende Schnittstelle für die am häufigsten vorkommenden Anwendungsfälle und eine komplexere Schnittstelle für Anwendungsfälle, für die die einfachere Schnittstelle nicht geeignet ist. Das Integrationsintervall wird vor Beginn der Integration festgelegt. Bei Verwendung der einfacheren Schnittstelle kehrt das entsprechende Unterprogramm nach Erreichen einer vorgegebenene Stelle t_{WANT} zum aufrufenden Anwendungsprogramm zurück. Bei Verwendung der komplexeren Schnittstelle erfolgt die Rückkehr zur Anwendung nach jedem Zeitschritt.

MATLAB-ODE-Suite. Die Mathematiksoftware MATLAB (siehe MATLAB-Homepage) enthält eine Reihe von Funktionen zur Lösung gewöhnlicher Differentialgleichungen (Shampine u. Reichelt 1997; Benker 2005). Darunter befinden sich auch zwei Implementierungen eingebetteter Runge-Kutta-Verfahren. Die Funktion `ode23` basiert auf dem Verfahren der Ordnung 3(2) von Bogacki u. Shampine (1989a). Die zweite Funktion, `ode45`, verwendet ein von Dormand u. Prince (1980) vorgeschlagenes Verfahren der Ordnung 5(4). Beide Funktionen verfügen über stetige Erweiterung und liefern eine grafische Darstellung der berechneten Lösung. Der Anwender kann entweder nur die Intervallgrenzen oder auch zusätzliche Zwischenpunkte vorgeben. Als Rückgabe liefert die Funktion die Approximationswerte an allen gewünschten Stellen des Integrationsintervalls. Die Lokalisierung von Ereignissen, d. h. die Bestimmung von Stellen, an denen eine vorgegebene Ereignisfunktion den Wert 0 annimmt, ist ebenfalls möglich. Wenn gewünscht, wird nach jedem Zeitschritt eine vom Anwender vorgegebene Ausgabefunktion aufgerufen.

2.1.4 Entwurfsentscheidungen für diese Arbeit

Da das Ziel dieser Arbeit darin besteht, das Speicherzugriffsverhalten von Implementierungen eingebetteter Runge-Kutta-Verfahren zu untersuchen und zu optimieren, ist es erforderlich, die zu erstellenden Implementierungen auf die wesentlichen Elemente zu reduzieren, welche in allen typischen Implementierungen vorkommen und ausschlaggebend für das Lokalitätsverhalten sind. Deshalb wurden folgende Entwurfsentscheidungen getroffen:

1. Es wird nur der Grundalgorithmus eines eingebetteten Verfahrens, bestehend aus der Berechnungsvorschrift (2.2) sowie einer Schrittweitenkontrolle, realisiert. Zusätzliche, nicht in jedem Fall benötigte Funktionalität, die über den Grundalgorithmus hinausgeht, wie etwa Steifheitserkennung, stetige Ausgabe, Abschätzung des globalen Fehlers und Ereignislokalisierung, wird nicht implementiert.

2. Möglichst alle Bestandteile des Verfahrens sollen innerhalb eines einzelnen Unterprogramms realisiert werden, um die Anzahl der Sprunganweisungen zu minimieren und so eine höhere Performance zu erreichen. Gleichzeitig wird dadurch ein besseres Lokalitätsverhalten bezüglich der Lesezugriffe auf das Maschinenprogramm bei dessen Ausführung erreicht.

3. Da das Differentialgleichungssystem kein Bestandteil des Verfahrens ist und die Möglichkeit bestehen soll, die Implementierungen mit Hilfe verschiedener Testprobleme zu untersuchen, wird die Auswertung der rechten Seite des Differentialgleichungssystems in einer separaten Funktion implementiert.

4. Damit eine Parallelisierung der Funktionsauswertung möglich wird und um zusätzliche Möglichkeiten für Schleifentransformationen zu erhalten, wird die Funktionsauswertung als skalare Funktion $f : \mathbb{N} \times \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}$ realisiert.

5. Um mehrere Testprobleme zu unterstützen, erfolgt der Funktionsaufruf für die Auswertung der rechten Seite über einen Funktionszeiger. Dadurch können die entsprechenden Funktionen der verschiedenen Testprobleme unterschiedliche Bezeichner erhalten, so daß der Wechsel zwischen Testproblemen ohne ein erneutes Binden des Maschinenprogramms möglich ist.

6. Auf die Verwendung einer kompensierten Summation sowie einer speziellen Behandlung des Intervallendes wird verzichtet, da dies über die zugrundeliegende Berechnungsvorschrift hinausgeht und viele der z. Z. verbreiteten Codes dies ebenfalls nicht unterstützen.

7. Auf eine Ausnahmebehandlung wird ebenfalls weitestgehend verzichtet. Dies führt zu einer Verbesserung der Performance und verringert den Codeumfang. Dadurch ist es möglich, das Laufzeit- und Lokalitätsverhalten der wesentlichen Berechnungsschritte unbeeinflußt von zusätzlichem Programmcode für Ausnahmebehandlungen zu untersuchen.

8. Statistische Informationen werden nur insoweit erfaßt, als es zur Untersuchung des Laufzeit- und Lokalitätsverhaltens notwendig ist. Um das Verhalten der Implementierungen unbeeinflußt von der Erfassung dieser Daten beobachten zu können, muß diese Funktionalität deaktivierbar sein.

Entsprechend dieser Anforderungen orientieren sich alle in dieser Arbeit vorgestellten Implementierungen an dem in Abb. 2.1 dargestellten Rahmenprogramm. Es besteht aus einer Funktion, die das Integrationsintervall $[t_0, t_e]$ mittels einer Schleife durchläuft. Der Körper dieser Schleife entspricht der Ausführung eines Zeitschrittes. Innerhalb jedes Zeitschrittes wird die Berechnungsvorschrift (2.2) angewandt und anschließend eine Schrittweitenkontrolle durchgeführt.

Der rechenintensivste Teil eines Zeitschrittes, im folgenden als Berechnungskernel oder Kernel bezeichnet, besteht aus der Umsetzung der Berechnungsvorschrift (2.2) und ist in den Programmzeilen 10 bis 16 dargestellt. Die in dieser Arbeit vorgestellten Implementierungen werden sich im wesentlichen darin unterscheiden, wie der Berechnungskernel realisiert wird. In den Abbildungen der jeweiligen Implementierungen wird deshalb auch nur der Berechnungskernel dargestellt werden.

Bemerkung. Im Sinne des mathematischen Verfahrens gehört die Berechnung der Skalierungsfaktoren $\mathbf{s}[i]$ in Zeile 16 des Rahmenprogramms näher zur Schrittweitenkontrolle als zur Umsetzung der Berechnungsvorschrift (2.2). Bei einer praktischen Implementierung wird die Berechnung der Skalierungsfaktoren jedoch oft in den Berechnungskernel integriert, da die Skalierungsfaktoren berechnet werden können, sobald der Vektor $\mathbf{v}_1$ bestimmt wurde. Bei einigen Implementierungen, welche die Vektoren $\mathbf{v}_1, \ldots, \mathbf{v}_s$ nicht explizit speichern, ist ein solches Vorgehen sogar erforderlich.

2.2 Allgemeine Ansätze zur Laufzeitverbesserung

Der folgende Abschnitt befaßt sich mit verschiedenen Faktoren, die Einfluß auf die Laufzeit eines Computerprogramms haben und stellt einige allgemeine Ansätze zur Laufzeitoptimierung von Computerprogrammen vor.

2.2.1 Wahl einer geeigneten Plattform

Die Laufzeit eines Computerprogramms wird vor allem durch die Leistungsfähigkeit der Plattform bestimmt, auf der das Programm ausgeführt wird. Da die Preisentwicklung im Bereich der Speichermodule es ermöglicht, daß moderne Workstations und Compute-Server mit einem relativ großen Hauptspeicher ausgestattet werden können, kann man davon ausgehen, daß die meisten Implementierungen eingebetteter Runge-Kutta-Verfahren ausschließlich im internen Hauptspeicher arbeiten können, ohne daß eine Auslagerung von Daten auf externe Speichergeräte, wie z. B. Festplatten, erforderlich ist.[2] Daher wird die Laufzeit seitens der Hardware hauptsächlich durch die Leistungsfähigkeit des Prozessors und des Speichersystems bestimmt.

[2] Eine Anwendungen mit sehr großem Speicherplatzbedarf ist z. B. die Lösung semi-diskretisierter partieller Differentialgleichungen. Für diesen Anwendungsfall werden z. T. spezielle Verfahren mit einem geringeren Speicherplatzbedarf gegenüber Standardverfahren eingesetzt, um eine Auslagerung zu vermeiden. Siehe dazu auch Abschnitt 4.2.4.

```
1:  void embedded_rk_integrator(double t_0, double t_e, double η[])
2:  {
3:      Allokiere Speicher für benötige Vektoren;
4:      Berechne Startwert für Schrittweite h;

5:      // Setzte t auf Anfang des Integrationsintervalls
6:      t := t_0;

7:      // Laufe über das Integrationsintervall, bis t_e erreicht ist
8:      while (t < t_e)
9:      {
10:         Berechne v_1, ..., v_s nach (2.2a);

11:         // Berechne Δη = η_{κ+1} − η_κ, vgl. (2.2b)
12:         Δη := h Σ_{l=1}^{s} b_l v_l;

13:         // Berechne Fehlervektor e = η̂_{κ+1} − η_{κ+1}, vgl. (2.2b) und (2.2c)
14:         e := h Σ_{l=1}^{s} b̃_l v_l;   // b̃_l = b̂_l − b_l

15:         // Berechne Skalierungsfaktoren
16:         s[j] := |η[j]| + |h · v_1[j]| für j := 1, ..., n;

17:         // Berechne Approximation des lokalen Fehlers
18:         ε := (1/TOL) max_{j:=1,...,n} |e[j]/s[j]|;

19:         if (ε ≤ 1.0)                                  // Zeitschritt wird akzeptiert
20:         {
21:             // Aktualisiere η und t
22:             η += Δη;
23:             t += h;

24:             // Bestimme Schrittweite für nächsten Zeitschritt
25:             h := h · max { 1/3, 0.9 · (1/ε)^{1/(p+1)} };
26:         }
27:         else                                          // Zeitschritt wird verworfen
28:         {
29:             // Bestimme Schrittweite für Wiederholung des Zeitschrittes
30:             h := h · max { 0.1, 0.9 · (1/ε)^{1/p} };
31:         }

32:         // Begrenze Schrittweite, um Intervallende nicht zu überschreiten
33:         h := min {h, t_e − t};
34:     }

35:     Gib nicht mehr benötigten Speicher frei;
36: }
```

Abb. 2.1: Rahmenprogramm eines sequentiellen eingebetteten Runge-Kutta-Verfahrens, an dem sich alle in dieser Arbeit vorgestellten Implementierungen orientieren.

Aktuell sind eine Vielzahl von Prozessortypen verbreitet, deren Architekturen unterschiedlichen Strategien folgen, um eine möglichst hohe Performance zu erreichen. So setzen beispielsweise einige Prozessortypen auf eine sehr hohe Taktfrequenz, während andere eher eine effizientere Ausnutzung zusätzlicher Funktionseinheiten anstreben. Die Verbesserung der Fertigungstechniken ermöglicht eine stetige Vergrößerung der Anzahl der pro Chip eingesetzten Bauelemente, wodurch ein großes Parallelitätspotential innerhalb *eines* Prozessors entsteht:

- **Datenparallelität:** Es werden Maschinenbefehle zur Verfügung gestellt, die mehrere Datenelemente gleichzeitig verarbeiten können. Ein Beispiel ist die gleichzeitige Ausführung der Additionen mehrerer Paare von Summanden. Funktionseinheiten, die eine derartige parallele Verarbeitung von Daten ermöglichen, werden als SIMD-Einheiten (für englisch: *single instruction multiple data*) bezeichnet (siehe z. B. Intel 2006b). Insbesondere Multimediaanwendungen und numerische Berechnungen, die häufig vektorbasierte Gleitkommaoperationen ausführen, können von solchen Funktionseinheiten profitieren. Es gibt sogar spezielle, meist in Supercomputern eingesetzte Vektorprozessoren, die für derartige Anwendungen entwickelt und optimiert wurden und ihr Potential aus der parallelen Verarbeitung vektorbasierter Daten schöpfen (z. B. Cray 2005; Kitagawa u. a. 2003; Kahle u. a. 2005).

- **Instruktionsparallelität:** Mehrere verschiedenartige Funktionseinheiten werden zur parallelen Ausführung von Operationen eines Instruktionsstromes genutzt. Dazu gibt es zwei konkurrierende Ansätze: (1) Ein Instruktionswort entspricht einem einzelnen Maschinenbefehl. Der Instruktionsscheduler des Prozessors analysiert dynamisch die Datenabhängigkeiten des Instruktionsstromes und verteilt die Operationen auf verfügbare Funktionseinheiten. Die Instruktionen können dabei in anderer Reihenfolge ausgeführt werden, als sie im Maschinenprogramm vorkommen (englisch: *out-of-order execution*, z. B. Intel 2006b). (2) Ein Instruktionswort enthält mehrere Maschinenbefehle, die parallel ausgeführt werden. Man spricht daher auch von VLIW-Prozessoren (für englisch: *very large instruction word*). Die Gruppierung der Maschinenbefehle erfolgt unter Beachtung der Datenabhängigkeiten statisch durch den Compiler (siehe z. B. Intel 2004).

- **Simultanes Multithreading (SMT):** Es werden mehrere Instruktionsströme (Threads) unterstützt, deren Instruktionen auf die verschiedenen Funktionseinheiten des Prozessors verteilt werden (siehe z. B. Tullsen u. a. 1995; Marr u. a. 2002; Kalla u. a. 2003; Mathis u. a. 2005; Kahle u. a. 2005; Kongetira u. a. 2005). Dazu wird für jeden Thread ein separater Registersatz zur Verfügung gestellt.

- **Chip-Multithreading (CMT)/Multi-Core-Prozessoren:** Auf einem Chip werden mehrere Prozessorkerne realisiert, die unabhängig voneinander jeweils einen oder – in Kombination mit SMT – mehrere Instruktionsströme ausführen können (siehe z. B. Intel 2006b; Sun 2004; Sinharoy u. a. 2005; Kahle u. a. 2005).

Aufgrund dieser Entwicklung der Prozessoren wird auch jedes in einer Hochsprache geschriebene „sequentielle“ Programm durch den Compiler oder den Instruktionsscheduler des Prozessors parallelisiert. Die anderen beiden Parallelitätsarten, simultanes Multithreading und Chip-Multithreading, erfordern dagegen eine explizite Aufsplittung des Programms in separate Kontrollflüsse durch den Programmierer.

Neben der reinen Ausführungsgeschwindigkeit des Prozessors und dessen Parallelitätspotential ist vor allem die Geschwindigkeit des Speichersystems von entscheidender Bedeutung für die Laufzeit eines Programms. Wichtige Maße für die Geschwindigkeit eines Speichers sind dessen *Speicherzugriffslatenz*, d. h. die maximale Wartezeit zwischen einer Datenanfrage und dem Beginn der Datenübertragung (Einheit ns oder Anzahl der Taktzyklen), und dessen *Speicherbandbreite*, d. h. die maximale Datenmenge, die pro Zeiteinheit übertragen werden kann (Einheit GB/s).

Da die Bereitstellung schneller Speicher nur bei kleinem Speichervolumen kostengünstig möglich ist, besitzen nahezu alle modernen Computersysteme eine Speicherhierarchie, bestehend aus mehreren Speichern unterschiedlicher Größe und Geschwindigkeit. Speicher, der sich auf einer höheren Ebene befindet, ist dabei schneller, aber auch kleiner als Speicher auf einer niedrigeren Ebene. Der Aufbau der Speicherhierarchie und die Strategie, nach der Daten zwischen den Hierarchieebenen ausgetauscht werden, hat demzufolge großen Einfluß auf die Ausführungsgeschwindigkeit eines Programms. Eine typische Speicherhierarchie aktueller Rechnersysteme besitzt fünf oder sechs Stufen. Auf der obersten Ebene befinden sich

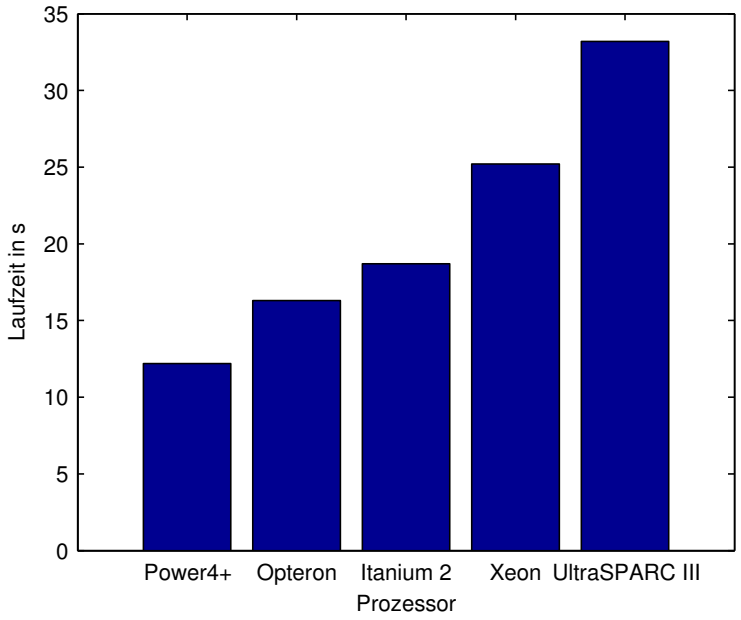


Abb. 2.2: Vergleich der Laufzeiten einer Implementierung eines eingebetteten Runge-Kutta-Verfahrens auf verschiedenen Prozessoren: IBM Power4+ (1.7 GHz), AMD Opteron DP 246 (2 GHz), Intel Itanium 2 (1,5 GHz), Intel Xeon MP (2,2 GHz) und UltraSPARC III Cu (1,05 GHz). Als Beispiel wurde Implementierung (A) (siehe Abschnitt 2.4) auf das Testproblem BRUSS2D-MIX (siehe Anhang B) angewendet.

die prozessorinternen Register. Die beiden untersten Ebenen bilden der Hauptspeicher und die Festplatten. Zwischen Prozessor und Hauptspeicher befindet sich ein meist zwei- oder dreistufiges Cache-System. Einige Beispiele für Cache-Hierarchien moderner Prozessoren zeigt Tab. 2.1.

Die Speicherhierarchien moderner Rechnersysteme arbeiten blockorientiert. Das bedeutet, daß Daten immer blockweise in die nächsthöhere oder nächsttiefere Hierarchieebene transferiert werden. Da die Latenzzeiten oft größer als die Blockübertragungszeiten sind, ist die Übertragung eines Blocks nur unwesentlich teurer als der Zugriff auf ein einzelnes Datum. Die blockweise Übertragung bietet jedoch den Vorteil, daß Speicheranfragen von Programmen, die oft auf räumlich benachbarte Daten zugreifen, häufiger aus höheren Hierarchieebenen befriedigt werden können. Programme, die das Potential eines Rechnersystems möglichst vollständig ausnutzen wollen, müssen deshalb so implementiert werden, daß sie von der Architektur der Speicherhierarchie und der blockweisen Übertragung profitieren (Meyer u. a. 2003).

Abbildung 2.2 zeigt ein Beispiel für einen Vergleich der Laufzeiten einer Implementierung eines eingebetteten Runge-Kutta-Verfahrens auf verschiedenen Prozessoren. Aus dieser Abbildung wird noch einmal deutlich, daß eine hohe Taktfrequenz allein nicht ausschlaggebend für eine geringe Laufzeit ist. So ist z. B. der untersuchte Xeon-Prozessor mit einer Taktfrequenz von 2,2 GHz deutlich langsamer als ein Power4+ mit 1,7 GHz, ein Opteron mit 2 GHz und ein Itanium 2 mit 1,5 GHz. Sogar ein UltraSPARC III, der nicht einmal halb so schnell getaktet ist wie der Xeon, kann mit einer nur etwa 1/3 höheren Laufzeit noch recht gut mit dem Xeon mithalten. Bei der Interpretation der Abbildung ist allerdings zu beachten, daß die ermittelten Laufzeiten nicht nur von dem verwendeten Prozessor, sondern auch von einer Vielzahl weiterer Faktoren abhängen. Einen großen Einfluß übt beispielsweise der verwendete Compiler aus, was im nächsten Abschnitt deutlich gemacht wird. Ausführliche Laufzeitexperimente mit den in dieser Arbeit betrachteten Implementierungsvarianten für unterschiedliche Testprobleme auf verschiedenen Prozessoren werden in den entsprechenden nachfolgenden Abschnitten präsentiert.

2.2.2 Wahl eines geeigneten Compilers

Bereits im vorigen Abschnitt wurde erwähnt, daß der Compiler, der in einer Hochsprache geschriebene Programme in ein ausführbares Maschinenprogramm übersetzt, einen Einfluß auf deren Ausführungsge-

Prozessor / Taktfrequenz	L1-Instruktionscache	L1-Datencache	L2-Cache	L3-Cache
Pentium 4 3,4 GHz (Intel, 2004)	Größe: 12 K µ-Operationen, Assoziativität: 8-fach	Größe: 16 KB, Assoziativität: 8-fach, Latenz: 4 Taktzyklen, Zeilengröße: 64 Byte	Größe: 1 MB, Assoziativität: 8-fach, Latenz: 18 Taktzyklen, Zeilengröße: 64 Byte	nicht vorhanden
Itanium 2 1,7 GHz (Intel, 2004)	Größe: 16 KB, Assoziativität: 4-fach, Latenz: 1 Taktzyklus, Zeilengröße: 64 Byte	Größe: 16 KB, Assoziativität: 4-fach, Latenz: 1 Taktzyklus, Zeilengröße: 64 Byte	Größe: 256 KB, Assoziativität: 8-fach, Latenz: 5 Taktzyklen, Zeilengröße: 128 Byte	Größe: 9 MB, Assoziativität: 18-fach (2 Wege pro MB), Latenz: 14 Taktzyklen, Zeilengröße: 128 Byte
Core Duo 2,33 GHz (Intel, 2006)	Größe: 32 KB, Assoziativität: 8-fach	Größe: 32 KB, Assoziativität: 8-fach, Latenz: 3 Taktzyklen, Zeilengröße: 64 Byte	Größe: 2 MB, Assoziativität: 8-fach, Latenz: 14 Taktzyklen, Zeilengröße: 64 Byte	nicht vorhanden
UltraSPARC III Cu 1,2 GHz (Sun, 2002)	Größe: 32 KB, Assoziativität: 4-fach, Latenz: 2 Taktzyklen, Zeilengröße: 32 Byte	Größe: 64 KB, Assoziativität: 4-fach, Latenz: 2 Taktzyklen, Zeilengröße: 32 Byte	Größe: 1, 4 oder 8 MB, Assoziativität: 2-fach, Latenz: ca. 19 Taktzyklen, Zeilengröße: 64, 256 oder 512 Byte	nicht vorhanden
Opteron DP 246 2,0 GHz (AMD, 2003)	Größe: 64 KB, Assoziativität: 2-fach, Zeilengröße: 64 Byte	Größe: 64 KB, Assoziativität: 2-fach, Latenz: 3 Taktzyklen, Zeilengröße: 64 Byte	Größe: 1 MB, Assoziativität: 16-fach, Latenz: ca. 20 Taktzyklen, Zeilengröße: 64 Byte	nicht vorhanden
Power4+ 1,7 GHz (IBM, 2001)	Größe: 64 KB, Assoziativität: direkt abbildend Zeilengröße: 128 Byte	Größe: 32 KB, Assoziativität: 2-fach, Latenz: ca. 4 Taktzyklen, Zeilengröße: 128 Byte	Größe: 1,5 MB/Chip, Assoziativität: 8-fach, Latenz: ca. 14 Taktzyklen, Zeilengröße: 128 Byte	extern, Größe: 32 MB/Chip, Assoziativität: 8-fach, Zeilengröße: 512 Byte

Tab. 2.1: Beispiele für Cache-Parameter aktueller Mikroprozessoren. Quellen: Intel (2006a), Intel (2004), Sun (2003a), AMD (2005), Delattre (2004), Tendler u. a. (2002), Behling u. a. (2001).

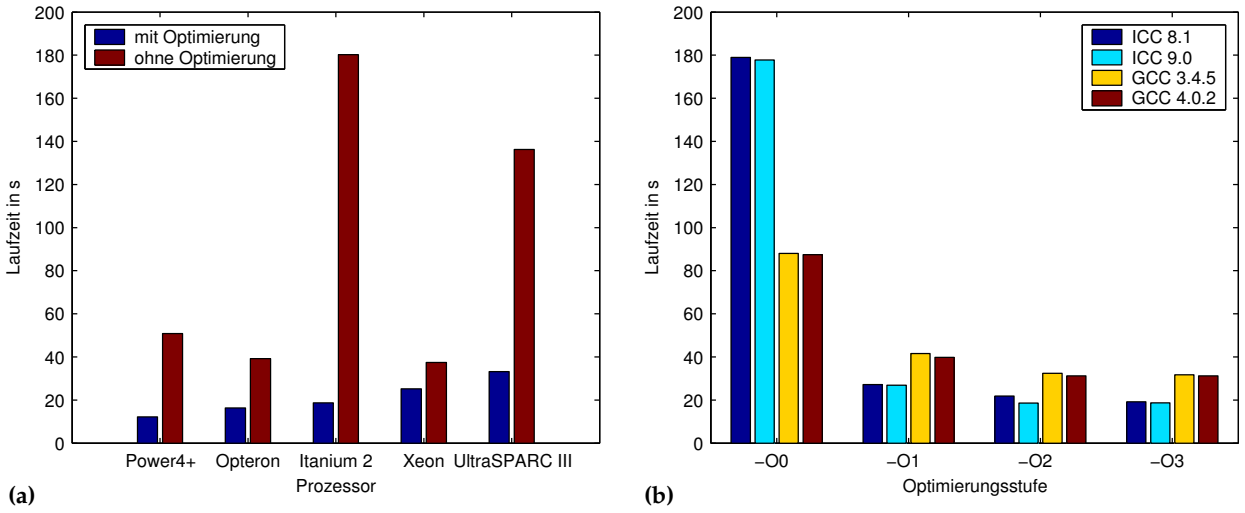


Abb. 2.3: Einfluß der Optimierung durch den Compiler auf die Laufzeit einer Implementierung eines eingebetteten Runge-Kutta-Verfahrens. **(a)** Laufzeiten auf verschiedenen Prozessoren wie in Abb. 2.2, jedoch sind hier zusätzlich die Laufzeiten bei Übersetzung mit ausgeschalteter Optimierung abgebildet. **(b)** Laufzeiten auf einem Intel-Itanium-2-Prozessor (1,5 GHz) für verschiedene Compiler (Intel C/C++ Compiler (ICC) 8.1 und 9.0, GNU Compiler Collection (GCC) 3.4.5 und 4.0.2) und verschiedene Optimierungsstufen. Als Beispiel wurde Implementierung (A) (siehe Abbschnitt 2.4) auf das Testproblem BRUSS2D-MIX (siehe Anhang B) angewendet.

schwindigkeit besitzt. Während dieser Übersetzung führt der Compiler eine Reihe von Optimierungen durch, die das Ziel haben, entweder die Laufzeit oder die Codegröße zu optimieren. Um dies zu erreichen, wurde im Bereich des Compilerbaus eine Vielzahl von Techniken entwickelt, die u. a. in den Büchern von Allen u. Kennedy (2002), Muchnick (1997), Polychronopoulos (1988) und Wolfe (1996) vorgestellt werden. Ein guter optimierender Compiler sollte versuchen:

- den Maschinenbefehlssatz der Zielplattform vollständig auszunutzen, um von speziellen Merkmalen der Zielplattform profitieren zu können (z. B. SIMD-Erweiterungen),
- die Anzahl der Maschinenbefehle zu minimieren, da pro Takt nur eine bestimmte maximale Anzahl von Befehlen ausgeführt werden kann,
- die Maschinenbefehle so anzuordnen, daß eine parallele Ausführung der Befehle auf verschiedenen Funktionseinheiten möglich ist,
- Speicherzugriffe durch effiziente Nutzung der prozessorinternen Register zu vermeiden und
- Programmtransformationen (z. B. für rechenzeitintensive Schleifen) durchzuführen, die das Lokalitätsverhalten verbessern (z. B. Tiling bzw. Blocking) oder eine parallele Ausführung ermöglichen.

Aufgrund der aktuellen Entwicklung im Bereich der Prozessorarchitektur ist es von zunehmender Bedeutung, wie gut der eingesetzte Compiler diese Punkte umsetzen kann. Hat man sich für eine Zielplattform entschieden, sollte man deshalb sorgfältig abwägen, welchen der verfügbaren Compiler man nutzen möchte. Da sich der Optimierungsprozeß oft durch die Übergabe spezieller Parameter beim Aufruf des Compilers beeinflussen läßt, müssen außerdem geeignete Parameter ausgewählt werden, um ein bestmögliches Ergebnis der Übersetzung zu erzielen.

Abbildung 2.3 zeigt den Einfluß der Optimierung durch den Compiler auf die Laufzeit einer Implementierung eines eingebetteten Runge-Kutta-Verfahrens. Besonders groß sind die Auswirkungen bei Verwendung des Intel-C/C++-Compilers auf einem Itanium-2-System. Aufgrund der VLIW-Architektur des Prozessors können hier die Funktionseinheiten nur dann effizient genutzt werden, wenn der Compiler die Maschinenbefehle auf geeignete Weise gruppiert.

2.2.3 Optimierung des Programmcodes

Zwar beherrschen moderne Compiler eine Vielzahl von Optimierungstechniken, dennoch hat der Programmierer den größten Einfluß darauf, wie schnell ein von ihm in einer Hochsprache formulierter Algorithmus ausgeführt wird. Um eine möglichst geringe Ausführungszeit zu erhalten, muß er das Programm so formulieren, daß es für den Compiler möglich ist, die unterstützten Optimierungstechniken anzuwenden. Wesentliche Optimierungen, die der Compiler nicht beherrscht oder aufgrund fehlender Informationen prinzipbedingt nicht durchführen kann, müssen manuell durchgeführt werden.

Will man die Laufzeit eines Programms minimieren, ist dies äquivalent zur Reduktion der Anzahl der Taktzyklen, die der Prozessor zu dessen Ausführung benötigt. Da pro Takt nur eine bestimmte maximale Anzahl von Befehlen ausgeführt werden kann, besteht ein Optimierungsansatz darin, die Anzahl der Maschinenbefehle des resultierenden Instruktionsstromes zu minimieren. Allerdings kann es bei der Abarbeitung eines Instruktionsstromes zu Wartezeiten kommen, z. B. weil der nächste auszuführende Maschinenbefehl nicht rechtzeitig geladen und dekodiert werden kann oder weil ein Maschinenbefehl Operanden benötigt, die nicht rechtzeitig geladen werden können. Diese Wartezeiten können sehr kurz sein, sie können aber beispielsweise auch mehrere 100 Takte betragen, z. B. wenn ein Operand aus dem Hauptspeicher geladen werden muß. Wurde das benötigte Datum auf ein externes Speichermedium ausgelagert, können die Wartezeiten sogar noch mehrere Größenordnungen darüber liegen. Für das Erreichen einer möglichst geringen Ausführungszeit ist die Minimierung derartiger Wartezeiten deshalb von besonders hoher Bedeutung.

Reduktion der Anzahl sequentiell ausgeführter Operationen

Die Anzahl der sequentiell ausgeführten Operationen, d. h. die maximale Pfadlänge des vom Instruktionsscheduler des Prozessors erzeugten Ablaufplans, läßt sich reduzieren, indem man entweder:

1. die Anzahl der Operationen verringert oder
2. die Operationen so anordnet, daß eine parallele Ausführung erfolgen kann.

Um die Anzahl der Operationen zu minimieren, ist es wichtig, bei der Formulierung des Algorithmus als Programm unnötige bzw. redundante Berechnungen zu vermeiden. Ein Beispiel dafür ist die Einsparung von Operationen bei der Berechnung eines Polynoms durch Anwendung des Horner-Schemas. Weitere mögliche Techniken zur Reduzierung der Operationszahl sind:

- Ausnutzung des Maschinenbefehlssatzes. Wenn der Zielprozessor einen Maschinenbefehl für eine zu lösende Aufgabe zur Verfügung stellt, sollte man diesen nutzen, anstatt die Funktionalität des Maschinenbefehls durch eine Programmfunktion nachzubilden (z. B. aufwendige Gleitkommaoperationen wie Wurzelberechnung oder Winkelfunktionen).
- Parallelisierung durch Verwendung von Operationen, die mehrere Daten gleichzeitig verarbeiten (SIMD-Operationen).
- Vermeidung von Funktionsaufrufen durch In-Line-Expansion, d. h. Einfügen des Funktionscodes an der Aufrufposition. Hierdurch kann man die Sprunganweisungen für den Funktionsaufruf und den Rücksprung einsparen sowie eine Reihe weiterer Anweisungen, die für die Parameterübergabe und das Sichern von Registern auf dem Stack erforderlich sind.
- Durchführung von Schleifentransformationen, wie sie im Bereich des Compilerbaus entwickelt werden, z. B. Vorausberechnung konstanter Ausdrücke, Aufrollen sehr kleiner, kompakter Schleifen oder Verschmelzen von Schleifen, um Sprung- und Vergleichsoperationen zu sparen.

Reduktion der Anzahl der Wartezyklen

Wartezyklen entstehen, wenn die Ausführung eines Maschinenbefehls verzögert werden muß. Eine Ursache dafür kann sein, daß der Befehl selbst noch nicht verfügbar ist. Eine zweite Ursache kann darin liegen, daß die Operanden eines Maschinenbefehls, d. h. die Daten, die dieser Befehl verarbeitet, nicht rechtzeitig verfügbar sind. Diese beiden möglichen Ursachen werden im folgenden näher betrachtet.

Wartezyklen für Operationen. Während eines Programmlaufes werden die Maschinenbefehle, aus denen das Programm besteht, in der durch das Programm vorgegebenen Ausführungsreihenfolge nacheinander in das Instruktionsregister des Prozessors geladen, dekodiert und ausgeführt. Die Gefahr, daß Maschinenbefehle nicht rechtzeitig zur Ausführung zur Verfügung stehen, liegt vor allem darin, daß ein Prozessor zwar theoretisch pro Takt einen oder sogar mehrere Befehle ausführen kann, das Laden eines Befehls bei einem Cache-Fehlzugriff aber eine große Zahl von Taktzyklen erfordert.

Die Zwischenspeicherung von Teilen des Programmcodes in einem Instruktionscache mit sehr geringer Zugriffszeit ist deshalb die Voraussetzung für eine effiziente Arbeitsweise eines Prozessors. Man profitiert dabei von der blockweisen Arbeitsweise der Speicherhierarchie, da – solange keine Sprunganweisungen vorkommen – die Befehle im Speicher in der Ausführungsreihenfolge abgelegt sind. Dadurch werden durch den Transfer eines Cacheblocks in der Regel immer mehrere aufeinanderfolgende Instruktionen in den Instruktionscache geladen. Moderne Prozessoren versuchen darüber hinaus, den Transfer zukünftig benötigter Befehle durch einen Prefetching-Mechanismus, kombiniert mit einer Sprungvorhersage, so früh wie möglich anzustoßen.

Die Anzahl der Wartezyklen für Operationen zu verringern, bedeutet, die Lokalität des Programmcodes zu optimieren. Hierzu gibt es z. B. folgende Ansätze:

- Mehrmalige Wiederverwendung von Codeblöcken, die sich im Cache befinden, z. B. mit Hilfe von Schleifen oder durch Verwendung von Unterprogrammen.

- Verringerung der Codegröße, damit möglichst der gesamte Programmcode in den Instruktionscache paßt. Dadurch kann man bei einer Wiederverwendung von Programmteilen (z. B. durch wiederholte Unterprogrammaufrufe oder Schleifen) die Anzahl der Zugriffe auf tiefere Ebenen der Speicherhierarchie minimieren. Außerdem sinkt die Gefahr von Interferenzen mit Daten in Caches auf tieferen Ebenen, die sowohl Instruktionen als auch Daten speichern (englisch: *unified caches*). Die wichtigste Technik zur Reduzierung der Codegröße ist die Wiederverwendung von Codeblöcken durch die Auslagerung in Unterprogramme oder die Verwendung von Schleifen.

- Linearisierung des Programmcodes durch die Vermeidung von (nichtvorhersagbaren) Sprunganweisungen. Damit das Instruktions-Prefetching optimal arbeiten kann, sollte man auf Sprunganweisungen möglichst verzichten oder den Programmcode so formulieren, daß die Sprungvorhersage des Prozessors das Sprungziel rechtzeitig ermitteln kann. Gegebenenfalls sollte man spezielle Maschinenbefehle nutzen, die die Sprungvorhersage unterstützen (z. B. durch Vorgabe des wahrscheinlicheren Ergebnisses eines Vergleichs bei einer bedingten Sprunganweisung).

- Vermeidung von Thrashing im Instruktionscache. Dies tritt auf, wenn im Wechsel verschiedene Programmbereiche angesprungen werden, die aufgrund ihrer Adresse und der Assoziativität des Caches auf gleiche Cachebereiche abgebildet werden.

Wartezyklen für Operanden. Für die Operanden, d. h. die zu verarbeitenden Daten, ergibt sich eine ähnliche Problematik wie für die Operationen. Allerdings ist der Speicherplatzbedarf der zu verarbeitenden Daten oft wesentlich höher als der des Programmcodes. Erschwerend hinzu kommt, daß die Zugriffsmuster, die durch Lese- und Schreibzugriffe auf Datenelemente erzeugt werden, in der Regel eine sehr unregelmäßige Struktur im Vergleich zum Zugriffsmuster auf den Programmcode besitzen. Es ist daher für einen Prozessorhersteller wesentlich schwieriger, eine effiziente Prefetching-Strategie für Datenzugriffe zu implementieren, weil zukünftige Zugriffe nur bedingt vorhersagbar sind. Deshalb liegt es in der Verantwortung des Programmierers, durch Optimierung der Lokalität der Datenzugriffe dafür zu sorgen, daß die Speicherhierarchie so gut wie möglich ausgenutzt wird. Gelingt dies nicht, führt dies auf modernen Prozessoren dazu, daß ein Großteil der verfügbaren Rechenleistung ungenutzt bleibt. Um dies zu vermeiden, kann man zur Optimierung der Lokalität der Datenzugriffe u. a. folgende Ansätze nutzen:

- Reduzierung der Anzahl der Speicherzugriffe durch effiziente Nutzung der verfügbaren Register. Dazu gehört auch die Vermeidung von Funktionsaufrufen, da diese in der Regel Speicherzugriffe implizieren, u. a. zum Speichern und Laden der Rücksprungadresse sowie zum Sichern von Registerinhalten.

- Zeitnahe Wiederverwendung im Cache befindlicher Daten. Wurde ein Block in den Cache transferiert, sollte man versuchen, den größtmöglichen Nutzen daraus zu ziehen, bevor er durch einen anderen Block verdrängt wird.
- Anordnung der Daten im Speicher, so daß man vom blockorientierten Transfer des Speichersystems profitieren kann. Wenn ein Block übertragen wird, dann sollte er möglichst viele Daten enthalten, die für sich zeitnah anschließende Berechnungen benötigt werden.
- Anordnung der Daten im Speicher, um bestmöglich vom Prefetching zu profitieren, falls der Zielprozessor einen solchen Hardwaremechanismus bietet. Sind spezielle Maschinenbefehle zur Realisierung eines Softwareprefetchings verfügbar, sollte man diese nutzen, um den Transfer von Blöcken rechtzeitig und gezielt anzustoßen, damit sie dann, wenn sie benötigt werden, ohne Wartezeit verfügbar sind.
- Vermeidung von Thrashing im Datencache, das auftreten kann, wenn im Wechsel auf Datenbereiche zugegriffen wird, die auf gleiche Cachebereiche abgebildet werden. Eine besondere Problematik in diesem Zusammenhang bildet der Speicherbereich für den Aufrufstack, in dem lokale Variablen von Funktionen sowie Übergabeparameter gespeichert werden. Ein allgemeiner Ansatz zur Minimierung der Cache-Fehlzugriffe, die dadurch entstehen, daß verschiedene Datenbereiche auf gleiche Cachebereiche abgebildet werden, wird z. Z. von Marco Höbbel (Universität Bayreuth) im Rahmen seiner Dissertation erforscht.
- Optimierung rechenzeitintensiver Schleifenstrukturen, die über große Datenfelder iterieren. Da Schleifen einen Codeblock wiederholt ausführen, besitzen sie ein hohes Potential zur Wiederverwendung von Daten. Sind die Datenstrukturen, auf die innerhalb der Schleife zugegriffen wird, größer als der Cache, besteht aber gleichzeitig auch die Gefahr, daß bereits in den Cache geladene Datenblöcke durch Zugriffe auf andere Datenblöcke aus dem Cache verdrängt werden und bei einem wiederholten Zugriff erneut aus einer tieferen Speicherhierarchieebene geladen werden müssen. Um dies zu vermeiden, kann man verschiedene Schleifentransformationstechniken (z. B. Vertauschen, Verschmelzen, Aufsplitten oder Blockung) anwenden, mit deren Hilfe sich die Größe der innerhalb der Schleifenstruktur verwendeten Datenmengen an die Gegebenheiten der Speicherhierarchie anpassen läßt.

2.2.4 Zusammenfassung und Schlußfolgerungen

Die Laufzeit eines Programms wird wesentlich durch die Zielplattform bestimmt, auf der es ausgeführt wird. Aufgrund der komplexen Architektur moderner Prozessoren, insbesondere aufgrund der umfangreichen Befehlssätze und des Trends zur Parallelverarbeitung von Daten und Instruktionen, sind für eine effiziente Nutzung der Leistungsfähigkeit eines Prozessors hochoptimierende Compiler erforderlich, die ein Maschinenprogramm erzeugen können, das durch den Instruktionsscheduler des Prozessors effizient auf dessen Funktionseinheiten verteilt werden kann.

Die Fähigkeiten eines Compilers zur Optimierung sind allerdings begrenzt, so daß eine große Verantwortung beim Programmierer verbleibt. Eine besondere Herausforderung ist dabei die effiziente Ausnutzung der Speicherhierarchie, da andernfalls die Programmausführung durch Wartezeiten verzögert wird. Wartezeiten können entstehen, wenn entweder

1. Operationen (Programmbefehle) oder
2. Operanden (Daten)

von einer niedrigen Ebene der Speicherhierarchie geladen werden müssen und deshalb nicht rechtzeitig zur Verfügung stehen.

Techniken zur Reduzierung dieser Wartezeiten werden allgemein als Optimierungen der **Lokalität der Speicherzugriffe** bezeichnet, da sie in der Regel darauf beruhen, die Speicherzugriffe entweder zeitlich (d. h. bezüglich der Zugriffsreihenfolge) oder räumlich (d. h. bezüglich der zugegriffenen Speicherposition) enger zusammenzufassen. Man unterscheidet deshalb:

- **Zeitliche Lokalität:** Auf ein und dieselbe Speicheradresse wird mehrfach zu zeitlich dicht aufeinanderfolgende Zeitpunkten zugegriffen.

- **Räumliche Lokalität:** Zu zeitlich dicht aufeinanderfolgenden Zeitpunkten wird auf räumlich eng beieinander liegende Speicheradressen zugegriffen.

Abhängig davon, ob die Anordnung der Speicherzugriffe auf Operationen oder Operanden optimiert werden soll, unterscheidet man weiterhin zwischen:

- **Lokalität des Programmcodes** und
- **Lokalität der Datenzugriffe**.

Auf beide Größen kann der Programmierer bei der Implementierung eines Programms Einfluß nehmen, indem er das Programm entsprechend strukturiert, geeignete Datenstrukturen auswählt und die Reihenfolge der Zugriffe auf diese Datenstrukturen optimiert. Der Compiler kann während der Übersetzung des Programms zusätzliche Optimierungen durchführen, indem er unter den verschiedenen Möglichkeiten zur Übersetzung des Programms eine möglichst gute auswählt und dazu eventuell Transformationen des Programmcodes durchführt. Er kann jedoch keinen Einfluß auf die Auswahl der Datenstrukturen nehmen oder gar Veränderungen des Algorithmus vornehmen, der dem Programm zugrunde liegt.

2.3 Implementierung eingebetteter Runge-Kutta-Verfahren

In Abschnitt 2.1 wurde beschrieben, was bei der Umsetzung der mathematischen Verfahrensvorschrift in ein Computerprogramm zu beachten ist, und es wurde ein Rahmenprogramm entwickelt, das als Ausgangspunkt für die in dieser Arbeit vorgestellten Implementierungsvarianten verwendet werden soll. In Abschnitt 2.2 wurden anschließend einige allgemeine Ansätze zur Optimierung von Computerprogrammen vorgestellt, wobei die besondere Bedeutung der Optimierung der Speicherzugriffslokalität hervorgehoben wurde. Im Anschluß daran betrachtet dieser Abschnitt spezielle Aspekte, die bei der Implementierung eingebetteter Runge-Kutta-Verfahren eine Rolle spielen, und diskutiert mögliche Auswirkungen auf die Laufzeit.

2.3.1 Direkte Umsetzung der Berechnungsvorschrift

Bevor jedoch näher auf verschiedene Implementierungsmöglichkeiten und spezielle Optimierungsansätze eingegangen wird, soll zunächst eine einfache Implementierungsvariante vorgestellt werden, die eine direkte Umsetzung der Berechnungsvorschrift (2.2) darstellt und als Basis für die nachfolgende Diskussion dienen wird.

Der erste Teil der Berechnungsvorschrift, Gleichung (2.2a), beschreibt die Berechnung der Stufenvektoren, $\mathbf{v}_1, \ldots, \mathbf{v}_s$:

$$\mathbf{v}_l = \mathbf{f}\left(t_\kappa + c_l h_\kappa, \eta_\kappa + h_\kappa \sum_{i=1}^{l-1} a_{li}\mathbf{v}_i\right) \quad \text{für} \quad l = 1, \ldots, s \ .$$

Da die Stufenvektoren $\mathbf{v}_1, \ldots, \mathbf{v}_s$ sequentiell voneinander abhängen, müssen sie mit aufsteigendem Index nacheinander berechnet werden. Dazu kann eine Schleife verwendet werden, die mit der Iterationsvariable l über die Stufen $1, \ldots, s$ iteriert. Zur Berechnung des zweiten Arguments der Funktion $\mathbf{f}$ führen wir die Argumentvektoren

$$\mathbf{w}_l = \eta_\kappa + h_\kappa \sum_{i=1}^{l-1} a_{li}\mathbf{v}_i \quad \text{für} \quad l = 1, \ldots, s \tag{2.3}$$

ein. Zur Berechnung von $\mathbf{w}_l$ muß ebenfalls eine Schleife verwendet werden, weil die Berechnungsvorschrift eine Summe mit variabler Grenze enthält. Da es in C keine Zuweisungsoperation für Vektoren gibt und in Abschnitt 2.1.4 festgelegt wurde, daß die Funktion $\mathbf{f}$ als skalare Funktion $f : \mathbb{N} \times \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}$ implementiert werden soll, müssen die Komponenten der Vektoren $\mathbf{w}_l$ und $\mathbf{v}_l$ einzeln mit Hilfe einer Schleife zugewiesen werden, die über die Dimension des Differentialgleichungssystems, n, iteriert.

Geht man zur Berechnung des Vektor $\Delta\eta$, des Fehlervektors $\mathbf{e}$ und des Skalierungsvektors $\mathbf{s}$ analog vor, ergibt sich der in Abb. 2.4 dargestellte Berechnungskernel. Ein solcher Kernel würde allerdings zu einer unnötig hohen Laufzeit führen, da er ein schlechtes Lokalitätsverhalten besitzt und ein großer Teil der

ausgeführten Operationen durch verschiedene Optimierungen eingespart werden könnten. Verschiedene Implementierungsmöglichkeiten und deren Auswirkungen auf das Lokalitätsverhalten und die Anzahl der Operationen werden im folgenden diskutiert.

2.3.2 Ansätze zur Reduzierung der Anzahl der ausgeführten Operationen

Durch einige einfache Veränderungen kann der Programmcode in Abb. 2.4 so angepaßt werden, daß die Anzahl der Operationen deutlich vermindert wird.

Abspaltung von Schleifeniterationen

Eine wichtige Technik in diesem Zusammenhang ist das Abspalten von Schleifeniterationen. Das bedeutet, daß man die ersten oder die letzten Iterationen einer Schleife aufrollt und vom Schleifenkörper trennt. Die Grenzen der Schleife müssen danach entsprechend angepaßt werden.

Beispiel 2.1. Von der Schleife

```
for (i := 1; i ≤ N; i++) work(i);
```

werden die ersten drei und die letzten zwei Iterationen abgespalten:

```
work(1);
work(2);
work(3);

for (i := 4; i ≤ N − 2; i++) work(i);

work(N − 1);
work(N);
```

Mit Hilfe dieser Technik ist es möglich, bei der Berechnung von Vektorsummen die Initialisierung der Summationsvektoren mit 0 zu vermeiden. Stattdessen wird die erste Iteration der Summationsschleife abgespalten, so daß das erste Glied der Summe zur Initialisierung verwendet werden kann. Auf diese Weise spart man für jede Vektorkomponente eine Zuweisungsoperation (z. B. das Initialisieren eines Registers mit 0, was in der Regel einen Prozessortakt benötigt). Dies kann bereits zu einem meßbaren absoluten Laufzeitgewinn führen, wenn man bedenkt, daß die Dimension des Differentialgleichungssystems unter Umständen sehr groß ein kann. Bei der in Abb. 2.4 dargestellten Implementierung ist der erzielbare Laufzeitgewinn jedoch noch wesentlich größer. Da hier die Initialisierung mit Hilfe einer separaten Schleife geschieht, die über die Vektorkomponenten iteriert, ist mit jeder Zuweisung auch ein Schreibzugriff auf das Speichersystem verbunden. Insbesondere dann, wenn die Dimension des Differentialgleichungssystems sehr groß ist, so daß bei der Iteration über einen Vektor der Cacheinhalt mehrfach überschrieben wird, könnten durch die Speicherzugriffe große Wartezeiten entstehen. Kann man diese Speicherzugriffe also vermeiden, führt dies oft zu einer signifikanten Verbesserung der Laufzeit. Hinzu kommt im Fall der Implementierung aus Abb. 2.4 noch, daß die Einsparung der Sprung- und Vergleichsoperationen, die zur Realisierung der Schleife nötig waren, zu einer weiteren Laufzeitverbesserung führt.

Multiplikationen mit der Schrittweite

An vier Stellen des Berechnungskernels in Abb. 2.4 finden Multiplikationen mit der Schrittweite h statt: Zur Berechnung der Argumentvektoren $\mathbf{w}_l$ (Zeile 7), zur Berechnung von $\Delta\eta$ (Zeile 14), zur Berechnung des Fehlervektors $\mathbf{e}$ (Zeile 19) und zur Berechnung der Skalierungsfaktoren (Zeile 21). Der zweite Faktor ist dabei immer entweder ein einzelner Stufenvektor oder eine zuvor berechnete gewichtete Summe von Stufenvektoren. Durch Ausnutzung der Distributivität läßt sich die Anzahl der Multiplikationen reduzieren, indem man die modifizierten Stufenvektoren

$$\tilde{\mathbf{v}}_l = h_\kappa \mathbf{v}_l \quad \text{für} \quad l = 1, \ldots, s \tag{2.4}$$

```
1:  // Berechne v_1, ..., v_s nach (2.2a)

2:  for (l := 1; l ≤ s; l++)
3:  {
4:      for (j := 1; j ≤ n; j++) w_l[j] := 0;

5:      for (i := 1; i < l; i++)
6:          for (j := 1; j ≤ n; j++) w_l[j] +=a_li v_i[j];

7:      for (j := 1; j ≤ n; j++) w_l[j] := η[j] + h w_l[j];

8:      for (j := 1; j ≤ n; j++) v_l[j] := f_j(t + c_l h, w_l);
9:  }

10: // Berechne Δη = η_{κ+1} − η_κ, vgl. (2.2b)

11: for (j := 1; j ≤ n; j++) Δη[j] := 0;

12: for (l := 1; l ≤ s; l++)
13:     for (j := 1; j ≤ n; j++) Δη[j] += b_l v_l[j];

14: for (j := 1; j ≤ n; j++) Δη[j] := hΔη[j];

15: // Berechne Fehlervektor e = η̂_{κ+1} − η_{κ+1}, vgl. (2.2b) und (2.2c)

16: for (j := 1; j ≤ n; j++) e[j] := 0;

17: for (l := 1; l ≤ s; l++)
18:     for (j := 1; j ≤ n; j++) e[j] += b̃_l v_l[j];   // b̃_l = b̂_l − b_l

19: for (j := 1; j ≤ n; j++) e[j] := he[j];

20: // Berechne Skalierungsfaktoren

21: for (j := 1; j ≤ n; j++) s[j] := |η[j]| + |h · v_1[j]|;
```

Abb. 2.4: Berechnungskernel eines eingebetteten Runge-Kutta-Verfahrens, realisiert als direkte vektororientierte Umsetzung der Berechnungsvorschrift (2.2).

einführt. Damit ergibt sich die modifizierte Berechnungsvorschrift

$$\tilde{\mathbf{v}}_l = h_\kappa \mathbf{f}\left(t_\kappa + c_l h_\kappa, \eta_\kappa + \sum_{i=1}^{l-1} a_{li}\tilde{\mathbf{v}}_i\right) \quad \text{für} \quad l = 1, \ldots, s \ , \tag{2.5a}$$

$$\eta_{\kappa+1} = \eta_\kappa + \sum_{l=1}^{s} b_l \tilde{\mathbf{v}}_l \ , \tag{2.5b}$$

$$\hat{\eta}_{\kappa+1} = \hat{\eta}_\kappa + \sum_{l=1}^{s} \hat{b}_l \tilde{\mathbf{v}}_l \ , \tag{2.5c}$$

mit den Skalierungsfaktoren

$$s_j = \left|\eta_{\kappa,j}\right| + \left|\tilde{v}_{1,j}\right| \quad \text{für} \quad j = 1, \ldots, n \ . \tag{2.6}$$

Die Multiplikation mit h erfolgt jetzt nur noch an einer Stelle, da direkt nach der Funktionsauswertung deren Ergebnis mit der Schrittweite multipliziert wird. Dadurch lassen sich nicht nur die eigentlichen Multiplikationsoperationen einsparen, sondern auch einige Schleifen, die nur zum Zweck der Multiplikation mit h implementiert wurden. Somit ist auch eine Verbesserung des Lokalitätsverhaltens möglich. Da die Reihenfolge der mathematischen Operationen verändert wurde, kann sich dieses Vorgehen allerdings auch auf den Rundungsfehler auswirken.

Reduzierung der Dimension von Feldzugriffen

Mehrdimensionale Feldzugriffe sind oft mit höheren Kosten verbunden als Speicherzugriffe über einen skalaren Zeiger, da die Adresse des Feldzugriffs erst berechnet werden muß. Wird das mehrdimensionale Feld als Block gespeichert, wie dies z. B. in FORTRAN der Fall ist, muß anhand der Indizes und der Größe des Typs der Datenelemente ein Offset ermittelt und zur Anfangsadresse des Blocks hinzuaddiert werden, um die Adresse des gewünschten Feldelements zu bestimmen. In C werden mehrdimensionale Feldzugriffe in der Regel über Zeigerfelder, d. h. durch indirekte Adressierung realisiert. Das bedeutet, daß für jede Dimension ein eindimensionaler Feldzugriff erfolgt. Es muß also für jede Dimension durch Multiplikation von Index und Größe des Datentyps ein Offset bestimmt und zur Anfangsadresse des betreffenden Feldes hinzuaddiert werden, damit anschließend durch eine Speicherzugriff die Anfangsadresse des Feldes der nächstniedrigeren Dimension geladen werden kann. Es besteht aber auch in C die Möglichkeit, z. B. mit Hilfe von Präprozessormakros, Feldzugriffe ähnlich wie in FORTRAN zu realisieren.

In jedem Fall ist ein mehrdimensionaler Feldzugriff in beiden Programmiersprachen relativ teuer, da entweder eine Reihe arithmetischer Operationen zur Bestimmung des Offsets oder mehrere Speicherzugriffe erforderlich sind. Dies kann sich insbesondere dann negativ auf die Laufzeit auswirken, wenn mehrdimensionale Feldzugriffe innerhalb von Schleifen verwendet werden. Man kann daher in diesem Fall oft eine Laufzeitverbesserung erreichen, indem man die Dimension der Feldzugriffe reduziert. Dies erreicht man, indem man die Anfangsadresse des Feldes der Dimension, über die iteriert wird, in einer Zeigervariablen speichert.

Beispiel 2.2. Die Reduktion der Dimension des mehrdimensionalen Zugriffs auf das Feld A im Programm

```
S := 0;
for (i := 1; i ≤ N; i++)
    for (j := 1; j ≤ N; j++)
        for (j := 1; j ≤ N; j++)
            S := S + A[i][j][k];
```

führt zu dem modifizierten Programm

```
S := 0;
for (i := 1; i ≤ N; i++)
{
    AI := A[i];
    for (j := 1; j ≤ N; j++)
    {
        AIJ := AI[j];
        for (j := 1; j ≤ N; j++)
            S := S + AIJ[k];
    }
}
```

Wurde das Feld A als Block der Größe N^3 allokiert, könnte man das Beispielprogramm auch wie folgt realisieren:

```
S := 0;
P := A[0][0];
for (i := 1; i ≤ N^3; i++) S := S + P[i];
```

Der erste im Beispiel gezeigte Transformationsschritt kann von vielen modernen Compilern, die die Optimierungstechnik des Verschiebens von schleifeninvariantem Code beherrschen, automatisch durchgeführt werden. Die zweite vorgeschlagene Transformation kann ein Compiler jedoch nicht automatisch durchführen, da er davon ausgehen muß, daß die Elemente des dreidimensionalen Feldes A in N^2 separaten Feldern der Größe N gespeichert werden.

Verschmelzen von Schleifen

Da für die Realisierung einer Schleife pro Iteration eine Sprung- und eine Vergleichsoperation erforderlich sind, kann man die Anzahl der ausgeführten Operationen reduzieren, indem man Schleifen mit gleichen Grenzen zu einer einzelnen Schleife verschmilzt. Dies hat allerdings auch zur Folge, daß sich das Lokalitätsverhalten verändert. Denn durch das Verschmelzen der Schleifen ensteht eine neue Schleife, deren Schleifenkörper im Vergleich zu den einzelnen Schleifen mehr Instruktionen enthält und in der Regel auch Speicherzugriffe auf eine größere Menge von Daten durchführt.

2.3.3 Lokalität des Programmcodes

Um Wartezeiten in bezug auf Operationen zu minimieren, ist es im allgemeinen wichtig, die Lokalität des Programmcodes zu optimieren, indem man z. B. die Codegröße reduziert, Codeblöcke wiederverwendet und unvorhersagbare Sprünge vermeidet. Im speziellen Fall der eingebetteten Runge-Kutta-Verfahren führt die einfache Berechnungsvorschrift dazu, daß alle in dieser Arbeit untersuchten Implementierungen eine sehr geringe Codegröße besitzen. Diese variiert zwar abhängig von der verwendeten Zielplattform und vom verwendeten Compiler, liegt jedoch für die betrachteten sequentiellen Implementierungen in der Regel unterhalb von 16 KB. Da viele moderne Prozessoren einen Instruktionscache mit einer Größe von mindestens 32 KB besitzen, kann daher der gesamte Programmcode einer Implementierung im Instruktionscache gespeichert werden, so daß sich ein nahezu optimales Lokalitätsverhalten bezüglich des Programmcodes ergibt.

Nur bei der Erzeugung von Programmcode für den Itanium-2-Prozessor mit Hilfe des Intel C/C++ Compilers wurden für viele Implementierungen Codegrößen oberhalb von 16 KB beobachtet. Auf anderen im Rahmen dieser Arbeit betrachteten Zielplattformen besitzen lediglich einige wenige Implementierungsvarianten eine Codegröße von über 16 KB, bei denen Schleifen aufgerollt wurden, wodurch die Codegröße angestiegen ist. Auch einige parallele Implementierungsvarianten führen aufgrund der zusätzlich notwendigen Synchronisations- bzw. Kommunikationsoperationen zu größerem Code. Ein 64 KB großer Instruktionscache genügt aber in der Regel, um auch diese Implementierungen aufnehmen zu können. Produktionscodes, die zusätzliche Funktionalität bieten müssen, können eine noch höhere Codegröße erfordern.

Allerdings ist aufgrund des derzeitigen Trends zu steigenden Cachegrößen davon auszugehen, daß bereits die nächste oder übernächste Prozessorgeneration ausreichend große Instruktionscaches besitzen wird, um selbst umfangreiche Produktionscodes aufnehmen zu können.

Sprünge werden innerhalb der Implementierungen hauptsächlich zur Realisierung von Schleifen sowie für Funktionsaufrufe verwendet. Die bedingten Sprunganweisungen zur Realisierung der Schleifen sind gut vorhersagbar, da in der überwiegenden Zahl der Fälle die Schleife wiederholt wird. Eine weitere bedingte Sprunganweisung wird für die Verzweigung innerhalb der Schrittweitenkontrolle benötigt, um in den beiden Fällen des Akzeptierens bzw. des Verwerfens eines Zeitschrittes unterschiedlichen Code auszuführen. Diese Verzweigung des Programmcodes ist für die Sprungvorhersageeinheit des Prozessors bzw. für den Compiler nur schwer vorhersehbar. Da in der überwiegenden Zahl der Fälle der Zeitschritt akzeptiert wird, wird sich eine gute Sprungvorhersage jedoch darauf einstellen. Wenn der Befehlssatz des Prozessors Befehle enthält, mit denen man das wahrscheinlichere Sprungziel festlegen kann, kann man auf diese Weise der Sprungvorhersage mitteilen, daß das Akzeptieren des Zeitschrittes häufiger vorkommt als das Verwerfen.

Funktionsaufrufe, etwa zur Auswertung der Funktion **f** oder bei Produktionscodes auch zur Ausgabe der Lösungsfunktion, können theoretisch zu Thrashing führen, wenn z. B. der Code der Funktionsauswertung und der Berechnungskernel auf gleiche Cachebereiche abgebildet werden. Da die meisten z. Z. eingesetzten Instruktionscaches bereits eine recht hohe Assoziativität besitzen, dürfte dies in der Praxis allerdings kaum auftreten.

2.3.4 Einfluß der Implementierung des Differentialgleichungssystems auf die Datenlokalität

Die Lokalität der Datenzugriffe hängt nicht ausschließlich von der Implementierung des Lösungsverfahrens ab, sondern wird auch von den Datenzugriffen aller vom Lösungsverfahren aufgerufenen Unterprogramme beeinflußt. Im Fall der in dieser Arbeit untersuchten Implementierungen ist dies ausschließlich die Funktionsauswertung der rechten Seite des Differentialgleichungssystems. Für das Erreichen einer geringen Laufzeit ist demzufolge auch eine geeignete Implementierung der Funktion **f** notwendig.

Zur Berechnung einer Komponente $f_j(t, \mathbf{w})$ wird entweder auf alle oder nur auf einen Teil der Komponenten von **w** zugegriffen. In jedem Fall ist es wichtig, daß durch die konkrete Implementierung und die Anordnung der Komponenten in einer günstigen Reihenfolge für eine hohe zeitliche und räumliche Lokalität der Zugriffe gesorgt wird. Wie sich eine hohe zeitliche Lokalität erreichen läßt, hängt stark vom zu realisierenden Differentialgleichungssystem ab und läßt sich nicht allgemeingültig beschreiben. Für das Erreichen einer guten räumlichen Lokalität sind die beiden folgenden Strategien hilfreich:

1. Wenn nur auf einen Teil der Komponenten von **w** zugegriffen wird, sollten diese möglichst nahe benachbart sein. Dazu muß gegebenenfalls die Reihenfolge der Komponenten verändert werden.
2. Unabhängig davon, wieviele Komponenten von **w** benutzt werden, sollte die Implementierung so erfolgen, daß die Speicherzugriffe nach (aufsteigenden) Adressen sortiert ausgeführt werden.

Ein Maß dafür, wie nahe benachbart die während der Funktionsauswertung benutzten Komponenten von **w** liegen, ist die Zugriffsdistanz der Funktion **f**:

Definition 2.1 (Zugriffsdistanz einer Komponentenfunktion)**.** Die *Zugriffsdistanz einer Komponentenfunktion* $f_j(t, \mathbf{w})$, bezeichnet als $d(f_j)$, ist der kleinste Wert b, so daß f_j nur die Teilmenge

$$\{w_{j-b}, w_{j-b+1}, \ldots, w_j, \ldots, w_{j+b-1}, w_{j+b}\}$$

der Komponenten des Argumentvektors **w** benutzt.

Definition 2.2 (Zugriffsdistanz einer Vektorfunktion)**.** Die *Zugriffsdistanz einer Vektorfunktion* **f**, bezeichnet als $d(\mathbf{f})$, ist die größte Zugriffsdistanz aller Komponentenfunktionen f_j der Funktion **f**, d. h.

$$d(\mathbf{f}) = \max_{1 \le j \le n} d(f_j) \ .$$

Dieses Maß wird nachfolgend eine wichtige Rolle bei der Analyse des Lokalitätsverhaltens von Implementierungsvarianten spielen. Es dient außerdem als Ausgangspunkt für eine Reihe von Optimierungen, die in Kapitel 4 vorgestellt werden.

2.3.5 Auswahl und Realisierung der Datenstrukturen

Voraussetzung für eine hohe Datenlokalität ist die Wahl geeigneter Datenstrukturen. Für die Implementierung eingebetteter Runge-Kutta-Verfahren spielen folgende Daten eine Rolle:

- die s Stufenvektoren $\mathbf{v}_1, \ldots, \mathbf{v}_s$ bzw. $\tilde{\mathbf{v}}_1, \ldots, \tilde{\mathbf{v}}_s$ der Dimension n,
- die s Argumentvektoren $\mathbf{w}_1, \ldots \mathbf{w}_s$ der Dimension n,
- der Approximationsvektor η der Dimension n,
- der Vektor $\Delta\eta$ der Dimension n,
- der Fehlervektor $\mathbf{e}$ der Dimension n,
- der Skalierungsvektor $\mathbf{s}$ der Dimension n,
- die Verfahrenskoeffizienten a_{ij} mit $1 \leq j < i \leq s$ sowie b_i, $\tilde{b}_i$ und c_i mit $1 \leq i \leq s$,
- Iterationsvariablen für Schleifen i, j, k, ...
- skalare Variablen für die Grenzen des Integrationsintervalls t_0 und t_e, die Zeit t und die Schrittweite h,
- die Approximation des lokalen Fehlers ϵ,
- gegebenenfalls zusätzliche Hilfsvektoren oder skalare Variablen.

Vor Beginn der Implementierung muß man sich die Frage stellen, welche von diesen Daten benötigt werden, ob eventuell zusätzliche Daten erforderlich sind und welche Datenstrukturen zur Speicherung der notwendigen Daten geeignet sind.

Ein Vektor, den man in jedem Fall benötigt, ist der Approximationsvektor η, da dieser die gesuchte Näherungslösung des Anfangswertproblems speichert. Auf den Vektor $\Delta\eta$ könnte man verzichten, wenn man den Approximationsvektor η erst aktualisiert, nachdem entschieden wurde, daß der aktuelle Zeitschritt akzeptiert wird. Der Fehlervektoren $\mathbf{e}$ und der Skalierungsvektor $\mathbf{s}$ sind ebenfalls nicht unbedingt erforderlich. Sie dienen lediglich zur Bestimmung der Approximation des lokalen Fehlers ϵ. Verschmilzt man die Schleifen zur Berechnung von $\mathbf{e}$, $\mathbf{s}$ und ϵ, könnte man auf die explizite Speicherung der beiden Vektoren verzichten und stattdessen ϵ schrittweise berechnen. Voraussetzung für die Einsparung der genannten Vektoren ist allerdings, daß alle Stufenvektoren (für den Skalierungsvektor $\mathbf{s}$ zumindest der erste Stufenvektor) gespeichert werden, was nicht in jeder Implementierung der Fall sein muß.

Die Implementierung in Abb. 2.4 speichert sowohl die Stufenvektoren $\mathbf{v}_1, \ldots, \mathbf{v}_s$ als auch die Argumentvektoren $\mathbf{w}_1, \ldots \mathbf{w}_s$. In der Regel genügt es jedoch, entweder nur die Stufenvektoren und einen Argumentvektor oder nur die Argumentvektoren zu speichern. Insbesondere bei einer großen Stufenzahl läßt sich so der Speicherbedarf nahezu halbieren. Betrachtet man die Implementierung in Abb. 2.4, so stellt man fest, daß die Argumentvektoren nicht mehr benötigt werden, nachdem sie für die Funktionsauswertung in Zeile 8 benutzt wurden. Es genügt also, für die Berechnung des Funktionsarguments einen einzelnen temporären Argumentvektor zu benutzen, der in der nächsten Stufe wiederverwendet werden kann. Speichert man alle Argumentvektoren, kann man auf die Speicherung der Stufenvektoren völlig verzichten. Dazu sind jedoch verschiedene Schleifentransformationen erforderlich, die alle im Berechnungskernel enthaltenen Schleifen miteinander verschmelzen. Beispiele für beide Fälle werden in Abschnitt 2.4 gezeigt.

Alle Vektoren, die von der Dimension des Differentialgleichungssystems abhängen, wird man in der Regel als dynamische Felder im Speicher ablegen, um verschiedene Testprobleme unterschiedlicher Dimension lösen zu können. Soll die Stufenanzahl ebenfalls variabel bleiben, muß man für die Speicherung der Stufen- bzw. Argumentvektoren auf ein zweidimensionales Feld zurückgreifen. Andernfalls, d. h. für festes s, könnte man auch s eindimensionaler Felder verwenden.

Ein dynamisches zweidimensionales Feld der Größe $s \times n$ kann man in C auf unterschiedliche Weise realisieren. Die übliche Vorgehensweise besteht darin, zunächst ein eindimensionales Zeigerfeld der Größe s zu allokieren. Für jedes Element dieses Feldes wird anschließend eine Speicheranforderung für ein Feld von n Gleitkommawerten ausgeführt, und dem entsprechenden Feldelement des Zeigerfeldes wird die Adresse des allokierten Feldes zugewiesen. Auf das j-te Element der i-ten Zeile eines Feldes A kann dann mit Hilfe

des Ausdrucks $A[i][j]$ zugegriffen werden. Bei diesem Vorgehen können die einzelnen Zeilen allerdings an beliebigen Stellen im Speicher liegen. Eine bedenkenswerte Alternative besteht daher darin, alle Zeilen in einem einzigen Block der Größe $s \times n$ zu speichern und die Elemente des Zeigerfeldes jeweils mit dem Anfang der zugehörigen Zeile zu initialisieren. Auf diese Weise erhält man sich die Zugriffsmöglichkeit über den Ausdruck $A[i][j]$. Die Umsetzung dieses Ausdrucks in ein Maschinenprogramm erfordert allerdings zusätzlich zur Speicheroperation zum Laden des gesuchten Datums eine zweite Speicheroperation zum Laden des Zeilenanfangs. Um dies zu umgehen, kann man ein Präprozessormakro, wie z. B.

#define FELDELEMENT(B, i, j) $\quad B[n \cdot i + j]$

verwenden, das anhand der Zeilen- und Spaltennummer den Offset relativ zur Anfangsadresse des eindimensionalen $s \times n$-Feldes B berechnet. Anstelle der Kosten des zusätzlichen Speicherzugriffs sind jetzt lediglich die in der Regel geringeren Kosten für die Multiplikation mit n zu tragen, vorausgesetzt, der Compiler hält den Wert von n in einem Register vor.

Für die Speicherung der Verfahrenskoeffizienten gibt es verschiedene Möglichkeiten, abhängig davon, welche Verfahren von der Implementierung unterstützt werden sollen. Um beliebige Verfahren unterstützen zu können, müssen dynamische Datenstrukturen, also z. B. ein- bzw. zweidimensionale Felder, verwendet werden, da die Größe der Datenstrukturen von der Stufenzahl s abhängig ist. Da für den Zugriff auf diese Datenstrukturen Speicheroperationen erforderlich sind, kann sich dies jedoch nachteilig auf die Laufzeit auswirken. Beschränkt man die Anzahl der Stufen auf einen festen maximalen Wert, kommen weitere Möglichkeiten der Speicherung in Betracht. So könnte man in diesem Fall z. B. statische Felder verwenden, die entweder als globale Variablen im Datensegment des Programms oder als lokale Variablen auf dem Aufrufstack gespeichert werden. Für den Zugriff sind jedoch auch in diesen beiden Fällen Speicheroperationen erforderlich. Weiterhin ist es möglich, die Schleifen über die Stufenanzahl aufzurollen. Speichert man die Verfahrenskoeffizienten in Feldern, kann man jetzt im Programmcode konstante Indizes verwenden. Alternativ ist es jetzt auch möglich, skalare Variablen für die Speicherung der Verfahrenskoeffizienten zu benutzen. Ein optimierender Compiler hat nun die Möglichkeit, diese Datenelemente in Registern abzulegen, wodurch ein effizienterer Zugriff möglich wird. Legt man sich auch auf ein bestimmtes Verfahren fest, kann man die Verfahrenskoeffizienten im Programm als Konstanten realisieren, indem man entweder bei der Deklaration der Variablen zusätzlich das Schlüsselwort **const** verwendet oder die Zahlenwerte der Koeffizienten direkt an der entsprechenden Programmstelle einfügt, an der sie benötigt werden. Der Compiler kann dadurch eventuell zusätzliche Optimierungen durchführen. Wenn z. B. der Prozessor eine spezielle Ladeoperation für das Laden einer Gleitkommakonstanten in ein Register unterstützt, können die Konstanten direkt im Codesegment des Programms als Operand einer solchen Operation gespeichert werden, wodurch keine Speicherzugriffe mehr erforderlich sind. In der Regel erfolgt jedoch das Speichern der Konstanten im Datensegment, und das Laden wird mittels einer Speicheroperation durchgeführt, so daß sich kein Vorteil gegenüber der Verwendung nicht-konstanter Variablen ergibt.

Skalare Variablen wie Iterationsvariablen für Schleifen, die Grenzen des Integrationsintervalls, die Zeit t, die Schrittweite h und die Approximation des lokalen Fehlers wird man in der Regel als lokale Variablen auf dem Aufrufstack speichern. Prinzipiell hat man natürlich auch die Möglichkeit, diese Variablen als globale Variablen im Datensegment oder als dynamische Datenstrukturen abzulegen. Wie bereits bei der Diskussion über die Speicherung der Verfahrenskoeffizienten deutlich wurde, resultieren daraus jedoch keine effizienteren Zugriffsmöglichkeiten.

Für die Reduzierung von Konfliktfehlzugriffen ist es wichtig, eine günstige Anordnung aller Datenstrukturen im Hauptspeicher zu finden. Konfliktfehlzugriffe können bei Verwendung mengenassoziativer Caches auftreten, wenn verschiedene mehrfach wiederverwendete Datenblöcke auf gleiche Cachebereiche abgebildet werden und die Assoziativität des Caches nicht groß genug ist, um alle diese Datenblöcke gleichzeitig speichern zu können. Um dies zu verhindern, ist es erforderlich, alle Datenstrukturen bezüglich ihrer Speicheradresse so auszurichten, daß die betroffenen Datenblöcke auf unterschiedliche Mengen innerhalb des Caches abgebildet werden. Ein Beispiel einer Implementierung eines eingebetteten Runge-Kutta-Verfahrens, deren Schleifenstruktur bei einer ungünstigen Ausrichtung der Datenstrukturen zu einer Häufung von Konfliktfehlzugriffen führt, wird in Abschnitt 2.4 diskutiert.

2.3.6 Berechnung der Stufen

Die Realisierung der Stufenberechnung, d. h. der schrittweisen Berechnung der Stufen- und Argumentvektoren, ist bestimmend für das Laufzeitverhalten der resultierenden Implementierung. Zur Berechnung der s Stufen muß in jeder Stufe l eine Auswertung der Funktion $\mathbf{f}$ erfolgen. Zuvor muß der Argumentvektor für diese Funktionsauswertung als gewichtete Summe der Stufenvektoren der vorhergehenden Stufen $1, \ldots, l-1$ berechnet worden sein. Infolgedessen ist die Anzahl der arithmetischen Operationen, die zur Berechnung der Stufen erforderlich sind, wesentlich größer als die Anzahl der für die übrigen Berechnungen des Zeitschrittes erforderlichen arithmetischen Operationen. Dies gilt ebenso für die Anzahl der Speicherzugriffsoperationen, da das Laden der Operanden der arithmetischen Operationen und das Schreiben des Resultats in nahezu allen Fällen eine Speicherzugriffsoperation erfordert. Aus diesem Grund ist es besonders wichtig, das Lokalitätsverhalten der Stufenberechnung zu optimieren, in dem man die Reihenfolge der Speicherzugriffsoperation unter Beachtung der Datenabhängigkeiten so verändert, daß die Anzahl der Cache-Fehlzugriffe minimiert wird.

Bei einer universellen Implementierung für beliebige Systemgrößen n und variable Stufenzahl s besteht die Berechnung der Stufen aus einer verschachtelten Schleifenstruktur. Da die Berechnung einer Stufe l erst gestartet werden kann, nachdem die Berechnung der Stufe $l-1$ abgeschlossen ist, iteriert dabei die äußerste Schleife immer über die Stufen $l = 1, \ldots, s$. Zur Berechnung des für die jeweilige Stufe benötigten Argumentvektors ist eine zweite Schleife erforderlich, die über alle zuvor berechneten Stufen $i = 1, \ldots, l-1$ oder alle nachfolgend zu berechnenden Stufen $i = l+1, \ldots, s$ läuft. Eine dritte Verschachtelungstiefe wird benötigt, um über die Komponenten des Differentialgleichungssystems $j = 1, \ldots, n$ zu iterieren.

Während die äußerste Schleife immer über die s Stufen laufen muß, können die inneren Schleifen durch Anwendung verschiedener Schleifentransformation so umgeformt werden, daß ein bestimmtes Lokalitätsverhalten resultiert. Abhängig davon, welche Datenstrukturen der Implementierung zugrunde liegen, sind so z. B. das Vertauschen, Aufsplitten oder das Zusammenfassen von Schleifen möglich. Auch Tiling- bzw. Blocking-Techniken können angewendet werden.

2.3.7 Berechnung der Hilfsvektoren

Neben der Stufenberechnung müssen einige weitere Vektoren der Dimension n berechnet werden. Dazu gehören der Vektor $\Delta\eta$, der Fehlervektor $\mathbf{e}$ und der Skalierungsvektor $\mathbf{s}$.

Der Skalierungsvektor $\mathbf{s}$ kann nach

$$s_j = \left|\eta_{\kappa,j}\right| + \left|h \cdot v_{1,j}\right| \quad \text{für} \quad j = 1, \ldots, n \ ,$$

bzw.

$$s_j = \left|\eta_{\kappa,j}\right| + \left|\tilde{v}_{1,j}\right| \quad \text{für} \quad j = 1, \ldots, n \ ,$$

berechnet werden, sobald der erste Stufenvektor, $\mathbf{v}_1$ bzw. $\tilde{\mathbf{v}}_1$ bekannt ist. Daher ist es möglich, die Berechnung des Skalierungsvektors mit der Schleife zur Berechnung des betreffenden Stufenvektors zu verschmelzen. Dies verbessert das Lokalitätsverhalten in der Hinsicht, daß die gerade berechnete Komponente des Stufenvektors und die ohnehin für die Berechnung dieser Komponente des Stufenvektors benötigte Komponente von η wiederverwendet werden können. Nachteilig für das Lokalitätsverhalten ist dabei allerdings, daß der Arbeitsraum (siehe Definition 2.3) der resultierenden Schleife größer ist als die einzelnen Arbeitsräume bei einer getrennten Berechnung des Stufenvektors und des Skalierungsvektors. Eine getrennte Berechnung dieser beiden Vektoren, wie in Abb. 2.4 dargestellt, kann daher in bestimmten Situationen vorteilhaft sein, etwa wenn der resultierende Arbeitsraum bei einer gemeinsamen Berechnung gerade größer wäre als eine der Cache-Stufen der Zielplattform. Wird eine andere Berechnungsvorschrift zur Bestimmung der Skalierungsfaktoren verwendet, müssen bei der Umsetzung der Berechnungsvorschrift die entsprechenden Datenabhängigkeiten berücksichtigt werden.

Der Vektor $\Delta\eta$ und der Fehlervektor $\mathbf{e}$ werden als gewichtete Summen aller Stufenvektoren

$$\Delta\eta = h_\kappa \sum_{l=1}^{s} b_l \mathbf{v}_l \qquad \text{bzw.} \qquad \Delta\eta = \sum_{l=1}^{s} b_l \tilde{\mathbf{v}}_l$$

und

$$\mathbf{e} = h_\kappa \sum_{l=1}^{s} \tilde{b}_l \mathbf{v}_l \qquad \text{bzw.} \qquad \mathbf{e} = \sum_{l=1}^{s} \tilde{b}_l \tilde{\mathbf{v}}_l$$

mit unterschiedlichen Gewichten b_l und $\tilde{b}_l$ berechnet. Hierzu ist jeweils eine zweifach verschachtelte Schleife erforderlich, um über die Stufen und die Systemdimension iterieren zu können. Die Reihenfolge der Verschachtelung ist dabei beliebig, beeinflußt jedoch das Lokalitätsverhalten.

Läßt man die äußere Schleife über die Stufen iterieren, so werden – stellt man sich diese äußere Schleife aufgerollt vor – s Schleifen ausgeführt, die elementweise über einen Stufenvektor, $\mathbf{v}$ oder $\tilde{\mathbf{v}}$, und einen Ergebnisvektor, $\Delta\eta$ oder $\mathbf{e}$, laufen. Daraus ergibt sich eine hohe räumliche Wiederverwendung. Ist die Systemdimension sehr groß, wird der Cache beim Durchlaufen der inneren Schleifen jedoch überschrieben, so daß die zeitliche Wiederverwendung der Elemente der Ergebnisvektoren innerhalb der äußeren Schleife nicht zu einem Cache-Treffer führt.

Läßt man jedoch die äußere Schleife über die Systemdimension laufen und iteriert mit der inneren Schleife über die Stufen, erreicht man eine hohe zeitliche Wiederverwendung der Elemente des Ergebnisvektors innerhalb der inneren Schleife. Eine gute Ausnutzung der räumlichen Lokalität ist ebenfall gegeben, vorausgesetzt, der Cache ist groß genug, um $s + 1$ Cachezeilen zu speichern. Weiterhin ist Voraussetzung, daß alle beteiligten Vektoren so im Speicher ausgerichtet sind, daß sie auf unterschiedliche Mengenadressen abgebildet werden.

Da sich die Berechnung der Vektoren $\Delta\eta$ und $\mathbf{e}$ nur durch die verwendeten Gewichte b_l und $\tilde{b}_l$ unterscheidet, ist es möglich, die Berechnung beider Vektoren in einer einzelnen Schleifenstruktur zusammenzufassen. Dies bietet den Vorteil einer höheren zeitlichen Lokalität, da Komponenten der Stufenvektoren direkt nacheinander für die Berechnung beider Ergebnisvektoren verwendet werden können. Realisiert man die Schleife über die Stufen als innere Schleife, genügt es, wenn der Cache $s + 2$ Cachezeilen aufnehmen kann, um sowohl eine hohe zeitliche als auch eine hohe räumliche Lokalität zu gewährleisten.

Neben der Realisierung einer separaten Schleifenstruktur zur Berechnung von $\Delta\eta$ und $\mathbf{e}$ besteht eine alternative Möglichkeit darin, die Berechnung dieser beiden Vektoren mit der Stufenberechnung zu verschmelzen. Die Stufenberechnung enthält eine äußere Schleife, die über die Stufen iteriert und in jeder Iteration einen Stufenvektor berechnet. Sobald eine Komponente eines Stufenvektors berechnet wurde, kann diese zur Berechnung von $\Delta\eta$ und $\mathbf{e}$ benutzt werden. Auf diese Weise verbessert sich die zeitliche Lokalität bezüglich der Lesezugriffe auf Stufenvektorkomponenten. Im Gegenzug verschlechtert sich allerdings die zeitliche Lokalität bezüglich der Schreibzugriffe auf die Vektoren $\Delta\eta$ und $\mathbf{e}$, da die Speicherzugriffsoperationen, die zur Berechnung der Stufen ausgeführt werden, mit diesen Schreibzugriffen interferieren können. Auch vergrößert sich durch die Schreibzugriffe auf $\Delta\eta$ und $\mathbf{e}$ der Arbeitsraum der Stufenberechnung.

Da zur Berechnung von $\Delta\eta$ und $\mathbf{e}$ alle Komponenten aller Stufenvektoren benötigt werden, ist die Berechnung innerhalb einer separaten Schleifenstruktur unabhängig von der Stufenberechnung nur dann möglich, wenn alle Stufenvektoren im Speicher gehalten werden. Sollen die Stufenvektorkomponenten nur temporär innerhalb der Stufenberechnung gespeichert werden, muß die Berechnung von $\Delta\eta$ und $\mathbf{e}$ gemeinsam mit der Berechnung der Stufen durchgeführt werden.

2.3.8 Optimierungen durch Spezialisierung

In Abschnitt 2.3.5 wurde bereits erwähnt, daß durch eine Spezialisierung, d. h. eine Festlegung bestimmter Parameter, zusätzliche Optimierungen ermöglicht werden. So erlaubt z. B. die Festlegung einer konstanten Stufenzahl oder die Festlegung auf bestimmte Verfahrenskoeffizienten eine veränderte Organisation der Datenstrukturen.

Bezüglich der Organisation des Programmcodes bedeutet eine feste Stufenzahl, daß alle Schleifen, die über die Stufen iterieren, eine feste Grenze besitzen. Dadurch wird das vollständige Aufrollen dieser Schleifen möglich, was wiederum zusätzliche Transformationen des Programmcodes erlaubt. Durch das Aufrollen der Schleifen können die zugehörigen Kontrollanweisungen, d. h. Vergleichs- und Sprungoperationen, eingespart werden. Weitere arithmetische Operationen können eingespart werden, wenn man sich auf ein bestimmtes Verfahren festlegt und dessen Verfahrenskoeffizienten viele Nullen und Einsen enthalten.

Das Aufrollen von Schleifen wirkt sich allerdings nachteilig auf die Größe des Programmcodes aus, da der Code des entsprechenden Schleifenkörpers mehrfach wiederholt wird. Insbesondere bei einer großen

Stufenzahl sollte deshalb darauf geachtet werden, daß die Codegröße des Integrators die Größe des Instruktionscaches möglichst nicht überschreitet. Andernfalls erhöht sich nicht nur die Wahrscheinlichkeit von Fehlzugriffen beim Laden der Operationen, auf tieferen Ebenen der Speicherhierarchie entsteht auch die Gefahr von Interferenzen mit Datenzugriffen.

2.4 Vergleich verschiedener Implementierungsvarianten

In diesem Abschnitt werden mehrere Implementierungsvarianten eingebetteter Runge-Kutta-Verfahren vorgestellt und miteinander verglichen, die sich im Rahmen der für diese Arbeit durchgeführten Untersuchungen als besonders effizient erwiesen haben.

2.4.1 Effiziente Implementierungsvarianten

Wie in Abschnitt 2.3 beschrieben, gibt es eine Vielzahl von Möglichkeiten zur Realisierung des Berechnungskernels. Sie unterscheiden sich anhand der verwendeten Datenstrukturen, jedoch vor allem anhand der Struktur der zugrundeliegenden Schleifen. Viele Implementierungsvarianten gehen daher durch Schleifentransformationen auseinander hervor bzw. können durch Schleifentransformationen hergeleitet werden.

Einige mögliche Transformationen wurden von Rauber u. Rünger (2001, 2004) vorgeschlagen. Im folgenden sollen vier Implementierungsvarianten vorgestellt werden, die unterschiedliche Strategien zur Optimierung des Laufzeit- und Lokalitätsverhaltens verfolgen. Dabei handelt es sich um zwei verfeinerte Versionen der von Rauber u. Rünger (2001, 2004) vorgeschlagenen Implementierungsvarianten sowie um zwei weitere mögliche Varianten. Allen Implementierungen ist gemeinsam, daß die Verfahrenskoeffizienten frei gewählt werden können.

Vektororientierte Implementierung

Die erste Implementierungsvariante, die vorgestellt werden soll, orientiert sich in ihrer Schleifenstruktur stark an dem in Abb. 2.4 dargestellten Berechnungskernel. Eine Implementierung mit weitgehend ähnlicher Schleifenstruktur wurde bereits von Rauber u. Rünger (2001, 2004) vorgestellt. Sie wurde von den Autoren mit dem Buchstaben (A) bezeichnet. Die in dieser Arbeit vorgestellte Implementierungsvariante erhält deshalb ebenfalls die Bezeichnung (A).

Die in dieser Implementierung realisierte Strategie zur Optimierung des Laufzeit- und Lokalitätsverhaltens besteht darin, die räumliche Lokalität möglichst optimal auszunutzen und gleichzeitig die Arbeitsräume der innersten Schleifen weitestgehend zu minimieren. Die in Abb. 2.4 dargestellte direkte Umsetzung der Berechnungsvorschrift erfüllt diese Anforderungen durch ihre vektororientierte Arbeitsweise bereits zu einem großen Teil. Der sich ergebende neue Berechnungskernel ist in Abb. 2.5 dargestellt. Er besitzt folgende Merkmale:

- Verwendete Vektoren sind:
  1. ein Approximationsvektor η,
  2. s Stufenvektoren $\mathbf{v}_1, \ldots, \mathbf{v}_s$,
  3. ein Argumentvektor $\mathbf{w}$ und
  4. die drei Hilfsvektoren $\Delta\eta$, $\mathbf{e}$ und $\mathbf{s}$.
- Der Berechnungskernel besteht aus drei Phasen:
  1. Berechnung der Stufen,
  2. Berechnung der Vektoren $\Delta\eta$ und $\mathbf{e}$ und
  3. Berechnung der Skalierungsfaktoren.
- Alle Schleifen, deren Schleifenkörper keine weitere Schleife enthalten, iterieren jeweils über die Systemdimension und benutzen dabei nur Komponenten von maximal drei Vektoren.

- In jeder Stufe l, $l \geq 2$, werden $l+1$ Schleifen ausgeführt, die auf diese Weise über die Systemdimension laufen. Für die erste Stufe wird nur eine einzige solche Schleife ausgeführt.
- Die Schleifen zur Berechnung der Vektoren $\Delta\eta$ und $\mathbf{e}$ wurden verschmolzen, um die zeitliche Lokalität bezüglich der Lesezugriffe auf die Stufenvektorelemente besser ausnutzen zu können.
- Bei allen Schleifen, die über Stufen iterieren, wurde die erste Iteration abgespalten, um unnötige Initialisierungen von Vektorelementen mit 0 zu vermeiden.

Die besonders gute Ausnutzung der räumlichen Lokalität ergibt sich daraus, daß alle Vektoren elementweise mit aufsteigendem Index durchlaufen werden. Somit werden direkt nach dem Zugriff auf das erste Element eines Cacheblocks auch alle anderen Elemente des Cacheblocks benutzt, solange das Ende des entsprechenden Vektors noch nicht erreicht ist. Ein weiterer Vorteil eines solchen Zugriffsmusters besteht darin, daß es auf Hardwareebene sehr gut vorhergesagt werden kann. Infolgedessen ist zu erwarten, daß auch auf Prozessoren mit einer sehr einfachen Prefetching-Einheit das Zugriffsmuster erkannt wird und der Transfer des nächsten benötigten Blocks frühzeitig gestartet werden kann. Die einfache Struktur der innersten Schleifen bietet außerdem den Vorteil, daß es für den Compiler leichter ist, Ansatzpunkte für eine Optimierung zu finden, z. B. ein an die Cachezeilengröße oder die Anzahl bestimmter Funktionseinheiten angepaßtes Aufrollen der Schleifen.

Der größte Nachteil der Implementierung (A) besteht darin, daß keine gute Ausnutzung der zeitlichen Lokalität mehr möglich ist, wenn die Größe der in einem Zeitschritt verwendeten Datenstrukturen die Größe des Caches überschreitet. Spätestens wenn die Größe der einzelnen Vektoren die Cachegröße erreicht, wird durch einen Lauf über einen Vektor der Cache mit den Daten des Vektors überschrieben, so daß alle zuvor darin enthaltenen Daten neu aus dem Speicher geladen werden müssen, wenn in nachfolgenden Schleifen darauf zugegriffen wird. Steigt die Größe der Vektoren weiter an, überschreibt ein Lauf über einen Vektor sogar dessen eigenen Anfangsbereich im Cache. Findet dann in der nachfolgenden Schleife erneut ein Lauf über diesen Vektor statt, muß der Anfangsbereich neu geladen werden, wodurch allerdings der Endbereich des Vektors aus dem Cache verdrängt wird. Infolgedessen führen nahezu alle zeitlichen Wiederverwendungen zu Kapazitätsfehlzugriffen. Im konkreten Fall der Implementierung (A) tritt diese Problematik auf, sobald die Größe der Vektoren ein Drittel bzw. die Hälfte der Cachegröße erreicht, da in den innersten Schleifen jeweils drei oder zwei Vektoren benutzt werden.

Ansätze zur Ausnutzung der zeitlichen Lokalität

Um die Berechnungen zu identifizieren, die hohes Potential für eine Ausnutzung der zeitlichen Lokalität bieten, betrachten wir ein Beispiel für einen Datenflußgraphen eines eingebetteten Runge-Kutta-Verfahrens (Abb. 2.6). Innerhalb des Datenflußgraphen lassen sich drei Bereiche erkennen, in denen eine häufige Wiederverwendung von Datenelementen stattfindet:

1. Die Komponenten der Stufenvektoren $\mathbf{v}_l$ bzw. $\tilde{\mathbf{v}}_l$, $l = 1, \ldots, s$, werden mehrfach gelesen, da sie zur Berechnung der entsprechenden Komponenten der Argumentvektoren $\mathbf{w}_i$, $i = l+1, \ldots, s$, benötigt werden.
2. Zur Berechnung der Komponenten der Argumentvektoren $\mathbf{w}_l$, $l = 2, \ldots, s$, werden die entsprechenden Komponenten der Stufenvektoren $\mathbf{v}_i$ bzw. $\tilde{\mathbf{v}}_i$, $i = 1, \ldots, l-1$, gewichtet aufsummiert. Dies führt in der Regel zu $l-1$ Schreibzugriffen auf die jeweilige Komponente von $\mathbf{w}_l$.
3. Zur Berechnung von $\eta_{\kappa+1}$ und $\hat{\eta}_{\kappa+1}$ (bzw. $\Delta\eta$ und $\mathbf{e}$) werden alle s Stufenvektoren gewichtet aufsummiert, was zu je s Schreibzugriffen auf die jeweilige Komponente des Ergebnisvektors führt.

Im folgenden werden zwei Implementierungsvarianten vorgestellt, die versuchen, dieses Potential auszunutzen.

Ausnutzung der zeitlichen Lokalität bezüglich der Schreibzugriffe

Die erste Implementierung, die das Ziel der Ausnutzung der zeitlichen Lokalität bezüglich der Schreibzugriffe auf die Argumentvektoren sowie die Vektoren $\Delta\eta$ und $\mathbf{e}$ verfolgt, wird mit dem Buchstaben (E) bezeichnet. Sie ist in Abb. 2.7 dargestellt und besitzt folgende Merkmale:

```
1:  // Berechne Stufenvektor v_1

2:  for (j := 1; j ≤ n; j++) v_1[j] := f_j(t + c_1 h, η);

3:  // Berechne Stufenvektoren v_2, ..., v_s

4:  for (l := 2; l ≤ s; l++)
5:  {
6:      for (j := 1; j ≤ n; j++) w[j] := a_{l1} v_1[j];

7:      for (i := 2; i < l; i++)
8:          for (j := 1; j ≤ n; j++) w[j] += a_{li} v_i[j];

9:      for (j := 1; j ≤ n; j++) w[j] := η[j] + h w[j];

10:     for (j := 1; j ≤ n; j++) v_l[j] := f_j(t + c_l h, w);
11: }

12: // Berechne Δη und Fehlervektor e

13: for (j := 1; j ≤ n; j++)
14: {
15:     Δη[j] := b_1 v_1[j];
16:     e[j] := b̃_1 v_1[j];
17: }

18: for (l := 2; l ≤ s; l++)
19:     for (j := 1; j ≤ n; j++)
20:     {
21:         Δη[j] += b_l v_l[j];
22:         e[j] += b̃_l v_l[j];
23:     }

24: // Berechne Skalierungsfaktoren

25: for (j := 1; j ≤ n; j++) s[j] := |η[j]| + |h · v_1[j]|;
```

Abb. 2.5: Implementierung (A).

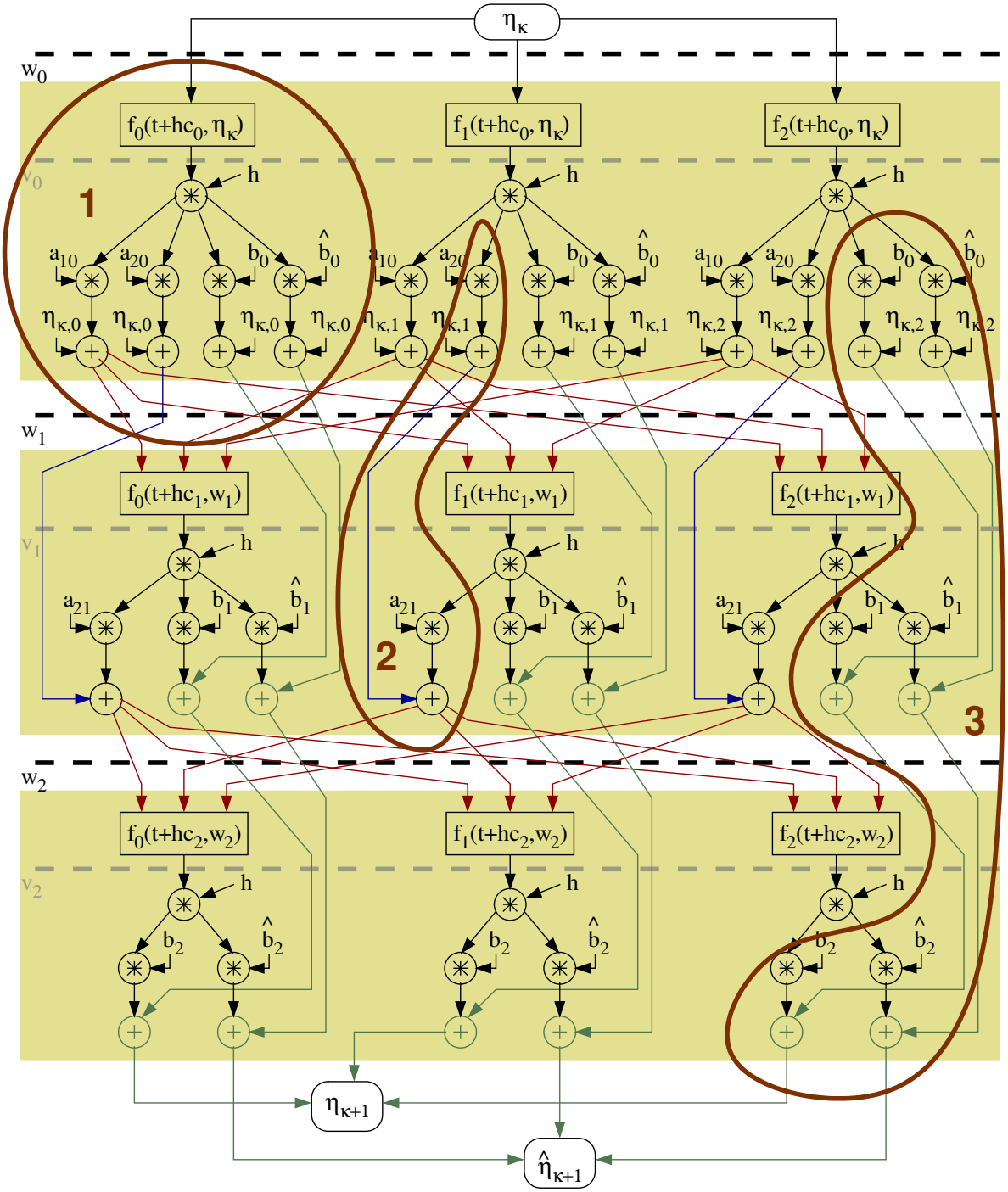


Abb. 2.6: Datenflußgraph eines eingebetteten Runge-Kutta-Verfahrens mit $s = 3$ Stufen für ein gewöhnliches Differentialgleichungssystem der Dimension $n = 3$. In Anlehnung an die Programmiersprache C beginnt die Numerierung aller Koeffizienten- und Vektorindizes bei 0. Gekennzeichnet sind die Berechnungen, die eine besonders hohe zeitliche Lokalität beinhalten: **(1)** Lesezugriffe auf Stufenvektorelemente, **(2)** Schreibzugriffe auf Argumentvektorelemente und **(3)** Schreibzugriffe auf $\eta_{\kappa+1}$ und $\hat{\eta}_{\kappa+1}$.

- Es werden die gleichen Vektoren wie in Implementierung (A) verwendet.
- Der Berechnungskernel besteht aus drei Phasen:
  1. Gemeinsame Berechnung der ersten Stufe und der Skalierungsfaktoren,
  2. Berechnung der Stufen $2, \ldots, s$ und
  3. Berechnung der Vektoren $\Delta\eta$ und $\mathbf{e}$.
- Wie in Implementierung (A) wurde bei allen Schleifen, die über Stufen iterieren, die erste Iteration abgespalten.
- Die Berechnung der Skalierungsfaktoren (Zeile 25 in Abb. 2.5) wurde mit der Schleife zur Berechnung des ersten Stufenvektors (Zeile 2 in Abb. 2.5) verschmolzen, um an dieser Stelle die zeitliche Lokalität bezüglich der Lesezugriffe auf diesen Stufenvektor ausnutzen zu können.
- Innerhalb der Schleife zur Berechnung der Stufen $2, \ldots, s$ wurden die beiden Schleifen in Zeile 7 und 8 der Abb. 2.5 vertauscht, so daß die innerste Schleife jetzt über die Stufen $i = 2, \ldots, l-1$ läuft. Weiterhin wurden alle Schleifen zur Berechnung des Argumentvektors $\mathbf{w}$, die über die Systemdimension iterieren (Zeile 6, 8 und 9 in Abb. 2.5), miteinander verschmolzen.
- Für die Speicherung der Zwischensummen zur Berechnung des Argumentvektors wird eine temporäre skalare Variable verwendet, deren Inhalt erst nach Abschluß der Summation an das entsprechende Vektorelement zugewiesen wird. Dadurch können Vektorzugriffe eingespart werden.
- Die Schleifen über die Systemdimension und über die Stufen zur Berechnung von $\Delta\eta$ und $\mathbf{e}$ (Zeile 18 und 19 in Abb. 2.5) wurden vertauscht und mit der in Zeile 13 der Abb. 2.5 beginnenden Schleife zusammengeführt. Für die Speicherung der Zwischensummen wird ebenfalls eine temporäre skalare Variable benutzt.

Ein effizienter Compiler kann diese Programmstruktur ausnutzen, indem er die temporären skalaren Variablen zur Berechnung der Komponenten der Argumentvektoren und der Vektoren $\Delta\eta$ und $\mathbf{e}$ während der jeweiligen Summation in einem Register vorhält und erst nach der endgültigen Berechnung das entsprechende Vektorelement mit einem einzigen Speicherzugriff zurückschreibt.

Die Programmstruktur der Implementierung (E) bietet aber auch eine gute Ausnutzung der räumlichen Lokalität. Sie benötigt dazu jedoch einen größeren Cache als Implementierung (A). Für die Ausnutzung der räumlichen Lokalität innerhalb der Schleife in Zeile 10 bis 15 der Abb. 2.7 müssen z. B. in Stufe s mindestens $s+1$ Cachezeilen für die Zwischenspeicherung des Argumentvektors $\mathbf{w}$, der Stufenvektoren $\mathbf{v}_1, \ldots, \mathbf{v}_{s-1}$ und des Approximationsvektors η verfügbar sein. Um Prefetching zu ermöglichen, müssen sogar mindestens $2(s+1)$ Cachezeilen verfügbar sein. Da die Vektoren $\Delta\eta$ und $\mathbf{e}$ gemeinsam berechnet werden, sollten für deren Berechnung $s+2$ bzw. $2(s+2)$ Cachezeilen verfügbar sein.

Verfügbarkeit bedeutet in diesem Zusammenhang, daß die Cachezeilen zusätzlich zum Speicherbedarf für die benutzten Verfahrenskoeffizienten und lokalen Variablen zur Verfügung stehen müssen. Wichtig ist in diesem Zusammenhang auch, daß die zugegriffenen Blöcke der benutzten Vektoren weitestmöglich auf unterschiedliche Cachemengen abgebildet werden, da es sonst zu unnötigen Konfliktfehlzugriffen kommen kann, wenn die Assoziativität des Caches nicht hoch genug ist.

Ausnutzung der zeitlichen Lokalität bezüglich der Lesezugriffe

Zur Ausnutzung der zeitlichen Lokalität bezüglich der Lesezugriffe auf die Stufenvektorkomponenten ist eine Veränderung der Datenstrukturen notwendig. Da die Stufenvektorkomponenten als Summanden in mehreren Summen (Berechnung der Argumentvektoren der nachfolgenden Stufen, Berechnung von $\Delta\eta$ und $\mathbf{e}$) vorkommen, müssen alle diese Zwischensummen im Speicher gehalten werden. Dies hat zur Folge, daß die Argumentvektoren $\mathbf{w}_2, \ldots, \mathbf{w}_s$ gespeichert werden müssen. Die Speicherung der Stufenvektoren selbst kann dagegen entfallen. Es ergibt sich daraus die in Abb. 2.8 gezeigte Implementierung, die in Anlehnung an Rauber u. Rünger (2001, 2004) mit dem Buchstaben (D) bezeichnet wird. Sie weist folgende Merkmale auf:

```
1:  // Berechne Stufenvektor v_1 und Skalierungsfaktoren

2:  for (j := 1; j ≤ n; j++)
3:  {
4:      v_1[j] := f_j(t + c_1 h, η);
5:      s[j] := |η[j]| + |h v_1[j]|;
6:  }

7:  // Berechne Stufenvektoren v_2, ..., v_s

8:  for (l := 2; l ≤ s; l++)
9:  {
10:     for (j := 1; j ≤ n; j++)
11:     {
12:         w := a_{l1} v_1[j];
13:         for (i := 2; i < l; i++) w += a_{li} v_i[j];
14:         w[j] := η[j] + hw;
15:     }

16:     for (j := 1; j ≤ n; j++) v_l[j] := f_j(t + c_l h, w);
17: }

18: // Berechne Δη und Fehlervektor e

19: for (j := 1; j ≤ n; j++)
20: {
21:     d := b_1 v_1[j];
22:     e := b̃_1 v_1[j];

23:     for (l := 2; l ≤ s; l++)
24:     {
25:         d += b_l v_l[j];
26:         e += b̃_l v_l[j];
27:     }

28:     Δη[j] := d;
29:     e[j] := e;
30: }
```

Abb. 2.7: Implementierung (E).

```
1:  // Berechne Stufenvektor v_1

2:  for (j := 1; j ≤ n; j++)
3:  {
4:      t̃ := h f_j(t + c_1 h, η);
5:      s[j] := |η[j]| + |t̃|;

6:      for (i := 2; i ≤ s; i++) w_i[j] := η[j] + a_i1 t̃;

7:      Δη[j] := b_1 t̃;
8:      e[j] := b̃_1 t̃;
9:  }

10: // Berechne Stufenvektoren v_2, ..., v_s

11: for (l := 2; l ≤ s; l++)
12:     for (j := 1; j ≤ n; j++)
13:     {
14:         t̃ := h f_j(t + c_l h, w_l);

15:         for (i := l + 1; i ≤ s; i++) w_i[j] += a_il t̃;

16:         Δη[j] += b_l t̃;
17:         e[j] += b̃_l t̃;
18:     }
```

Abb. 2.8: Implementierung (D).

- Verwendete Vektoren sind:
  1. ein Approximationsvektor η,
  2. $s-1$ Argumentvektoren $\mathbf{w}_2, \ldots, \mathbf{w}_s$ und
  3. die drei Hilfsvektoren $\Delta\eta$, $\mathbf{e}$ und $\mathbf{s}$.
- Der Berechnungskernel besteht aus zwei Phasen:
  1. Berechnung der ersten Stufe, Berechnung der Skalierungsfaktoren und Initialisierung von $\Delta\eta$ und $\mathbf{e}$,
  2. Berechnung der Stufen $2, \ldots, s$ und der Vektoren $\Delta\eta$ und $\mathbf{e}$.
- In jeder Stufe l erfolgt ein Lauf über die Systemdimension.
- Während dieses Laufes wird für jede Komponente j die rechte Seite des Differentialgleichungssystems ausgewertet und das Ergebnis mit der Schrittweite h multipliziert. Dies entspricht der Berechnung der modifizierten Stufenvektorkomponente $\tilde{v} = \tilde{\mathbf{v}}_{lj}$. Dieser Wert wird nacheinander auf die Argumentvektoren $\mathbf{w}_{l+1}, \ldots, \mathbf{w}_s$ und die Vektoren $\Delta\eta$ und $\mathbf{e}$ aufaddiert, nachdem mit dem jeweiligen Gewicht multipliziert wurde.
- Um dies zu realisieren, laufen die innersten Schleifen über die Stufen $l+1, \ldots, s$.

Im Vergleich zu den Implementierungen (A) und (E) benötigt Implementierung (D) weniger Speicherplatz, da zwei Vektoren weniger gespeichert werden müssen. Neben der Speicherplatzersparnis ist der wichtigste Vorteil dieser Implementierung jedoch die zeitlich nahe Wiederverwendung der Stufenvektorkomponenten, die es sogar ermöglicht, die Stufenvektorkomponenten nur temporär in einem Register zu speichern und auf Speicherzugriffe bezüglich der Stufenvektorkomponenten völlig zu verzichten. In der

ersten Stufe werden die Stufenvektorkomponenten zur Berechnung des Skalierungsvektors $\mathbf{s}$, der $s-1$ Argumentvektoren $\mathbf{w}_2, \ldots, \mathbf{w}_s$ und der Vektoren $\Delta\eta$ und $\mathbf{e}$ insgesamt $(s+2)$-fach wiederverwendet. In jeder nachfolgenden Stufe l werden die Stufenvektorkomponenten jeweils $(s-l+2)$-fach widerverwendet, da hier die Berechnung des Skalierungsvektors entfällt.

Um die räumliche Lokalität ausnutzen zu können, müssen für jeden der innerhalb der Schleifen über die Systemdimension (Abb. 2.8, Zeile 2–9 und Zeile 12–18) zugegriffenen Vektoren mindestens eine oder besser zwei Cachezeilen verfügbar sein. Für die Schleife in Zeile 2–9 bedeutet dies, daß mindestens $s+3$ bzw. $2(s+3)$ Cachezeilen verfügbar sein sollten. Für die Schleife in Zeile 12–18 müssen mindestens $s-l+3$ bzw. $2(s-l+3)$ Zeilen verfügbar sein. Da in beiden Schleifen aber auch die Funktionsauswertung aufgerufen wird, reichen diese Cachezeilen allein jedoch nicht aus, um die Ausnutzung der räumlichen Lokalität zu ermöglichen. Es muß zusätzlich genügend Platz innerhalb des Caches verfügbar sein, um die von der Funktionsauswertung referenzierten Daten aufnehmen zu können.

Schleifen-Tiling

Eine häufig verwendete Technik zur Optimierung verschachtelter Schleifenstrukturen ist das sogenannte *Tiling* bzw. *Blocking* (von englisch: *tile* bzw. *block*). Darunter versteht man eine Schleifentransformation, die die Tiefe einer Schleifenstruktur erhöht, so daß im Ergebnis die inneren Schleifen (Tile-Schleifen) nur noch über kleine, bezüglich der Indexmenge lokale Teilbereiche des Iterationsraumes der ursprünglichen Schleifenstruktur iterieren. Die Tile-Größe, d. h. der Indexbereich der Tile-Schleifen, sollte dabei so gewählt werden, daß die innerhalb der Tile-Schleifen zugegriffenen Daten im Cache gespeichert und somit in nachfolgenden inneren Schleifen oder nachfolgenden Iterationen der äußeren Schleifen wiederverwendet werden können.

Im Fall der eingebetteten Runge-Kutta-Verfahren kann Schleifen-Tiling auf die verschachtelte Schleifenstruktur innerhalb der Stufenberechnung und auf die Berechnung der Vektoren $\Delta\eta$ und $\mathbf{e}$ angewendet werden. Für das Implementierungsbeispiel gehen wir von Implementierung (D) aus. Die resultierende Implementierung nennen wir (Dblock). Ziel der Anwendung des Tilings ist es, eine Schleifenstruktur zu erzeugen, bei der die innersten Schleifen wie bei Implementierung (A) über einen Teilbereich der Systemdimension laufen und somit eine gute Ausnutzung der räumlichen Lokalität ermöglichen, wobei die Teilbereiche jedoch hinreichend klein gewählt werden, um eine zeitnahe Wiederverwendung zu ermöglichen, bevor es zu Kapazitätsfehlzugriffen kommt.

Um dies zu erreichen, werden zunächst die Schleifen in Zeile 2 und 12 der Abb. 2.8, die über die Systemdimension laufen, durch eine verschachtelte Schleife der Form

```
for (j := 1; j ≤ n; j += B)
    for (k := 0; k < min(B, n − j + 1); k++)
```

ersetzt. Der Wert B wird dabei als Tile- oder Blockgröße bezeichnet. Anschließend wird der Schleifenkörper der k-Schleife aufgespalten, so daß jede darin enthaltene Anweisung oder Schleife nun innerhalb einer separaten k-Schleife ausgeführt wird. Dazu ist es erforderlich, einen Vektor $\hat{\mathbf{v}}$ der Dimension k zur Speicherung der modifizierten Stufenvektorkomponenten einzuführen, da die skalare Variable $\tilde{v}$ nicht mehr ausreichend ist. Als letzter Schritt wird die innerste Schleife über die Stufen $l+1, \ldots, s$ in Zeile 15 (bzw. 6) der Abb. 2.8 mit der umschließenden k-Schleife vertauscht. Es ergibt sich die in Abb. 2.9 dargestellte Implementierung.

2.4.2 Theoretische Analyse des Lokalitätsverhaltens

Um das Lokalitätsverhalten der vorgestellten Implementierungsvarianten vergleichen zu können, soll zunächst eine theoretische Analyse der Schleifenstrukturen durchgeführt werden.

Speicherplatzbedarf

Eine erste wichtige Größe mit Einfluß auf das Lokalitätsverhalten ist der Speicherplatzbedarf der Implementierungen, d. h. die akkumulierte Größe aller Datenstrukturen, die innerhalb des Berechnungskernels verwendet werden. Bei der Beschreibung der Implementierungen in Abschnitt 2.4.1 wurde bereits genannt,

```
1:  // Berechne Stufenvektor v_1

2:  for (j := 1; j ≤ n; j += B)
3:  {
4:      for (k := 0; k < min(B, n − j + 1); k++) v̂[k + 1] := h f_{j+k}(t + c_1 h, η);
5:      for (k := 0; k < min(B, n − j + 1); k++) s[j + k] := |η[j + k]| + |v̂[k + 1]|;

6:      for (i := 2; i ≤ s; i++)
7:          for (k := 0; k < min(B, n − j + 1); k++) w_i[j + k] := η[j + k] + a_{i1} v̂[k + 1];

8:      for (k := 0; k < min(B, n − j + 1); k++) Δη[j + k] := b_1 v̂[k + 1];
9:      for (k := 0; k < min(B, n − j + 1); k++) e[j + k] := b̃_1 v̂[k + 1];
10: }

11: // Berechne Stufenvektoren v_2, ..., v_s

12: for (l := 2; l ≤ s; l++)
13:     for (j := 1; j ≤ n; j += B)
14:     {
15:         for (k := 0; k < min(B, n − j + 1); k++) v̂[k + 1] := h f_{j+k}(t + c_l h, w_l);

16:         for (i := l + 1; i ≤ s; i++)
17:             for (k := 0; k < min(B, n − j + 1); k++) w_i[j + k] += a_{il} v̂[k + 1];

18:         for (k := 0; k < min(B, n − j + 1); k++) Δη[j + k] += b_l v̂[k + 1];
19:         for (k := 0; k < min(B, n − j + 1); k++) e[j + k] += b̃_l v̂[k + 1];
20:     }
```

Abb. 2.9: Implementierung (Dblock).

welche Vektoren in den jeweiligen Implementierungen verwendet werden. Unter Vernachlässigung des geringen Speicherplatzbedarfs für die Verfahrenskoeffizienten sowie für die verwendeten skalaren Variablen ergibt sich der Speicherplatzbedarf S der Implementierungen als Funktion der Stufenzahl s und der Systemgröße n und im Fall der Implementierung (Dblock) zusätzlich als Funktion der Blockgröße B. Eine Übersicht der in den Implementierungen verwendeten Vektoren und des sich daraus ergebenden Speicherplatzverbrauchs ist in Tab. 2.2 dargestellt.

Implementierung	verwendete Vektoren	$S(s, n[, B])$
(A), (E)	η, $\Delta\eta$, $\mathbf{e}$, $\mathbf{s}$, $\mathbf{v}_1, \ldots, \mathbf{v}_s$, $\mathbf{w}$	$(s+5)n$
(D)	η, $\Delta\eta$, $\mathbf{e}$, $\mathbf{s}$, $\mathbf{w}_2, \ldots, \mathbf{w}_s$	$(s+3)n$
(Dblock)	η, $\Delta\eta$, $\mathbf{e}$, $\mathbf{s}$, $\mathbf{w}_2, \ldots, \mathbf{w}_s$, $\hat{\mathbf{v}}$	$(s+3)n + B$

Tab. 2.2: Speicherplatzbedarf der allgemeinen sequentiellen Implementierungen.

Arbeitsräume der Schleifen

Unter einem Arbeitsraum soll folgendes verstanden werden:

Definition 2.3 (Arbeitsraum)**.** Der *Arbeitsraum* (englisch: *working space*) eines Programmabschnittes ist die Menge aller innerhalb dieses Programmabschnittes referenzierten Datenelemente.

Aus diesem Blickwinkel betrachtet, stellt der Speicherplatzbedarf des Berechnungskernels gleichzeitig dessen Arbeitsraum dar. Sofern innerhalb der Schrittweitenkontrolle keine zusätzlichen Datenelemente re-

ferenziert werden, entspricht der Arbeitsraum des Berechnungskernels dem Arbeitsraum eines Zeitschrittes und damit dem Arbeitsraum der Gesamtimplementierung.

Für eine genauere Analyse des Lokalitätsverhaltens ist es sinnvoll, die Arbeitsräume der einzelnen Schleifen zu betrachten, aus denen die Berechnungskernel der Implementierungen bestehen. Dazu werden für jede Schleife die drei folgenden Mengen bestimmt:

R_I: Menge der pro Iterationsschritt referenzierten Datenelemente.

R_L: Gesamtmenge der durch die Ausführung aller Iterationen der Schleife referenzierten Datenelemente.

I: Menge der Werte, die die Iterationsvariable während der Ausführung der Schleife annimmt.

Weiterhin wird für jede dieser Mengen die Anzahl der darin enthaltenen Datenelemente $|R_I|$, $|R_L|$ und $|I|$ ermittelt.

Zur Bestimmung der Arbeitsräume der in einem Programm enthaltenen Schleifen beginnt man mit den innersten Schleifen, da man hier die Datenreferenzen direkt ablesen kann. Nachdem die Arbeitsräume der innersten Schleifen bekannt sind, kann man diese für die Bestimmung der Arbeitsräume der umschließenden Schleifen verwenden. Auf diese Weise wird fortgefahren, bis die oberste Verschachtelungsebene erreicht ist.

Für die Analyse der eingebetteten Runge-Kutta-Verfahren werden zur Vereinfachung nur die Zugriffe auf Elemente der wichtigsten Vektoren berücksichtigt. Dies kann dadurch gerechtfertigt werden, daß die verbleibenden Daten, d. h. Verfahrenskoeffizienten und skalare Variablen, sehr häufig benutzt werden, aber nur wenig Speicherplatz benötigen. Diese Daten werden sich daher nahezu immer im Cache befinden oder sogar in Registern vorgehalten werden, weshalb keine spezielle Lokalitätsoptimierung für diese Daten erforderlich ist. Werden die Daten im Cache gehalten, nehmen sie aufgrund ihrer geringen Größe meist nur einen kleinen Teil des Caches ein.

Beispiel 2.3. Jedes s-stufige eingebettete Runge-Kutta-Verfahren verwendet maximal $s(s-1)/2+3s$ Verfahrenskoeffizienten. Für ein Verfahren der Ordnung 3(2) mit $s=3$ Stufen werden demzufolge 12 Verfahrenskoeffizienten benötigt, die einen Speicherbedarf von 96 Byte besitzen. Ein Verfahren der Ordnung 8(7) mit $s=13$ Stufen benötigt 117 Verfahrenskoeffizienten und besitzt dementsprechend einen Speicherbedarf von 936 Byte.

Nur für sehr kleine Caches muß daher der Speicherbedarf der Verfahrenskoeffizienten explizit berücksichtigt werden. Geht man davon aus, daß sich die Verfahrenskoeffizienten immer im Cache befinden, kann man dies auf einfache Weise nachholen, indem man den Gesamtspeicherbedarf der Verfahrenskoeffizienten zu den Arbeitsräumen aller Schleifen hinzuzählt.

Eine Schwierigkeit bei der Analyse eingebetteter Runge-Kutta-Verfahren ergibt sich daraus, daß ein Teil des Programmcodes – der Code zur Auswertung der rechten Seite des Differentialgleichungssystems – problemabhängig ist und sein Lokalitätsverhalten daher nur schwierig durch einen allgemeinen Ansatz beschrieben werden kann. Prinzipiell ist es erlaubt, daß zur Auswertung der rechten Seite beliebige, aus Sicht der Funktionsauswertung interne Datenstrukturen verwendet werden. Da in vielen Fällen jedoch keine internen Datenstrukturen benötigt werden oder diese nur einen geringen Speicherplatz erfordern, beschränken wir uns in dieser Arbeit für die Analyse auf die Zugriffe auf den Argumentvektor.

Im allgemeinen muß man davon ausgehen, daß durch die Funktionsauswertung alle n Komponenten des Argumentvektors benutzt werden. Oft wird jedoch tatsächlich nur ein wesentlich kleinerer Teil der Komponenten benutzt, deren Anzahl sich unter Verwendung der Zugriffsdistanz $d(\mathbf{f})$ der Funktion $\mathbf{f}$ nach oben abschätzen läßt. Für die Bestimmung der Arbeitsräume der in den Implementierungen verwendeten Schleifen wird deshalb angenommen, daß jeder Aufruf einer Komponentenfunktion f_j lesend auf $D = 2d(\mathbf{f})+1$ Komponenten des Argumentvektors zugreift.

Die ermittelten Arbeitsräume der Implementierungen (A), (E), (D) und (Dblock) sind in Tab. 2.3 aufgelistet. Für diese Aufstellung wurde im Fall der Implementierung (Dblock) vereinfachend angenommen, daß die Systemgröße n ein ganzzahliges Vielfaches der Blockgröße B ist, so daß die k-Schleifen immer genau B Iterationen ausführen. Wenn n kein ganzzahliges Vielfaches von B ist, wird dadurch der Arbeitsraum R_L der k-Schleifen und der Arbeitsraum R_I der j-Schleifen für eine Iteration der j-Schleife zu groß geschätzt werden. Auf alle übrigen $\lfloor n/B \rfloor$ Iterationen der j-Schleife hat dies jedoch keinen Einfluß.

Zeile	I	$\lvert I\rvert$	R_I	R_L	$\lvert R_I\rvert$	$\lvert R_L\rvert$
Implementierung (A)						
2	$j=1,\ldots,n$	n	$\mathbf{v}_1[j]$; D Komp. von η	$\mathbf{v}_1, \eta$	$D+1$	$2n$
4	$l=2,\ldots,s$	$s-1$	$\mathbf{w}, \mathbf{v}_1, \ldots, \mathbf{v}_l, \eta$	$\mathbf{w}, \mathbf{v}_1, \ldots, \mathbf{v}_s, \eta$	$(l+2)n$	$(s+2)n$
6	$j=1,\ldots,n$	n	$\mathbf{w}[j], \mathbf{v}_1[j]$	$\mathbf{w}, \mathbf{v}_1$	2	$2n$
7	$i=2,\ldots,l-1$	$l-2$	$\mathbf{w}, \mathbf{v}_i$	$\mathbf{w}, \mathbf{v}_2, \ldots, \mathbf{v}_{l-1}$	$2n$	$(l-1)n$
8	$j=1,\ldots,n$	n	$\mathbf{w}[j], \mathbf{v}_i[j]$	$\mathbf{w}, \mathbf{v}_i$	2	$2n$
9	$j=1,\ldots,n$	n	$\mathbf{w}[j], \eta[j]$	$\mathbf{w}, \eta$	2	$2n$
10	$j=1,\ldots,n$	n	$\mathbf{v}_l[j]$; D Komp. von $\mathbf{w}$	$\mathbf{w}, \mathbf{v}_l$	$D+1$	$2n$
13	$j=1,\ldots,n$	n	$\mathbf{v}_1[j], \Delta\eta[j], \mathbf{e}[j]$	$\mathbf{v}_1, \Delta\eta, \mathbf{e}$	3	$3n$
18	$l=2,\ldots,s$	$s-1$	$\mathbf{v}_l, \Delta\eta, \mathbf{e}$	$\mathbf{v}_2, \ldots, \mathbf{v}_s, \Delta\eta, \mathbf{e}$	$3n$	$(s+1)n$
19	$j=1,\ldots,n$	n	$\mathbf{v}_l[j], \Delta\eta[j], \mathbf{e}[j]$	$\mathbf{v}_l, \Delta\eta, \mathbf{e}$	3	$3n$
25	$j=1,\ldots,n$	n	$\mathbf{v}_1[j], \mathbf{s}[j], \eta[j]$	$\mathbf{v}_1, \mathbf{s}, \eta$	3	$3n$
Implementierung (E)						
2	$j=1,\ldots,n$	n	$\mathbf{v}_1[j], \mathbf{s}[j]$; D Komp. von η	$\mathbf{v}_1, \eta, \mathbf{s}$	$D+2$	$3n$
8	$l=2,\ldots,s$	$s-1$	$\mathbf{w}, \mathbf{v}_1, \ldots, \mathbf{v}_l, \eta$	$\mathbf{w}, \mathbf{v}_1, \ldots, \mathbf{v}_s, \eta$	$(l+2)n$	$(s+2)n$
10	$j=1,\ldots,n$	n	$\mathbf{w}[j], \mathbf{v}_1[j], \ldots, \mathbf{v}_{l-1}[j], \eta[j]$	$\mathbf{w}, \mathbf{v}_1, \ldots, \mathbf{v}_{l-1}, \eta$	$l+1$	$(l+1)n$
13	$i=2,\ldots,l-1$	$l-2$	$\mathbf{v}_i[j]$	$\mathbf{v}_2[j], \ldots, \mathbf{v}_{l-1}[j]$	1	$l-2$
16	$j=1,\ldots,n$	n	$\mathbf{v}_l[j]$; D Komp. von $\mathbf{w}$	$\mathbf{w}, \mathbf{v}_l$	$D+1$	$2n$
19	$j=1,\ldots,n$	n	$\mathbf{v}_1[j], \ldots, \mathbf{v}_s[j], \Delta\eta[j], \mathbf{e}[j]$	$\mathbf{v}_1, \ldots, \mathbf{v}_s, \Delta\eta, \mathbf{e}$	$s+2$	$(s+2)n$
23	$l=2,\ldots,s$	$s-1$	$\mathbf{v}_l[j]$	$\mathbf{v}_2[j], \ldots, \mathbf{v}_s[j]$	1	$s-1$
Implementierung (D)						
2	$j=1,\ldots,n$	n	$\mathbf{w}_2[j], \ldots, \mathbf{w}_s[j], \Delta\eta[j], \mathbf{e}[j], \mathbf{s}[j]$; D Komp. von η	$\mathbf{w}_2, \ldots, \mathbf{w}_s, \eta, \Delta\eta, \mathbf{e}, \mathbf{s}$	$D+s+2$	$(s+3)n$
6	$i=2,\ldots,s$	$s-1$	$\mathbf{w}_i[j], \eta[j]$	$\mathbf{w}_2[j], \ldots, \mathbf{w}_s[j], \eta[j]$	2	s
11	$l=2,\ldots,s$	$s-1$	$\mathbf{w}_l, \ldots, \mathbf{w}_s, \Delta\eta, \mathbf{e}$	$\mathbf{w}_2, \ldots, \mathbf{w}_s, \Delta\eta, \mathbf{e}$	$(s-l+3)n$	$(s+1)n$
12	$j=1,\ldots,n$	n	$\mathbf{w}_{l+1}[j], \ldots, \mathbf{w}_s[j], \Delta\eta[j], \mathbf{e}[j]$; D Komp. von $\mathbf{w}_l$	$\mathbf{w}_l, \ldots, \mathbf{w}_s, \Delta\eta, \mathbf{e}$	$D+s-l+2$	$(s-l+3)n$
15	$i=l+1,\ldots,s$	$s-l$	$\mathbf{w}_i[j]$	$\mathbf{w}_{l+1}[j], \ldots, \mathbf{w}_s[j]$	1	$s-l$
Implementierung (Dblock)						
2	$j=1,B,\ldots,n$	$\lceil n/B\rceil$	$\hat{\mathbf{v}}$; B Komp. von $\mathbf{w}_2, \ldots, \mathbf{w}_s, \Delta\eta, \mathbf{e}, \mathbf{s}$; $B+D-1$ Komp. von η	$\hat{\mathbf{v}}, \mathbf{w}_2, \ldots, \mathbf{w}_s, \eta, \Delta\eta, \mathbf{e}, \mathbf{s}$	$D+(s+4)B-1$	$(s+3)n+B$
4	$k=0,\ldots,B-1$	B	$\hat{\mathbf{v}}[k+1]$; D Komp. von η	$\hat{\mathbf{v}}$; $B+D-1$ Komp. von η	$D+1$	$2B+D-1$
5	$k=0,\ldots,B-1$	B	$\hat{\mathbf{v}}[k+1], \eta[j+k], \mathbf{s}[j+k]$	$\hat{\mathbf{v}}$; B Komp. von η, $\mathbf{s}$	3	$3B$
6	$i=2,\ldots,s$	$s-1$	$\hat{\mathbf{v}}$; B Komp. von $\mathbf{w}_i, \eta$	$\hat{\mathbf{v}}$; B Komp. von $\mathbf{w}_2, \ldots, \mathbf{w}_s, \eta$	$3B$	$(s+1)B$
7	$k=0,\ldots,B-1$	B	$\hat{\mathbf{v}}[k+1], \mathbf{w}_i[j+k], \eta[j+k]$	$\hat{\mathbf{v}}$; B Komp. von $\mathbf{w}_i, \eta$	3	$3B$
8	$k=0,\ldots,B-1$	B	$\hat{\mathbf{v}}[k+1], \Delta\eta[j+k]$	$\hat{\mathbf{v}}$; B Komp. von $\Delta\eta$	2	$2B$
9	$k=0,\ldots,B-1$	B	$\hat{\mathbf{v}}[k+1], \mathbf{e}[j+k]$	$\hat{\mathbf{v}}$; B Komp. von $\mathbf{e}$	2	$2B$
12	$l=2,\ldots,s$	$s-1$	$\hat{\mathbf{v}}, \mathbf{w}_l, \ldots, \mathbf{w}_s, \Delta\eta, \mathbf{e}$	$\hat{\mathbf{v}}, \mathbf{w}_2, \ldots, \mathbf{w}_s, \Delta\eta, \mathbf{e}$	$(s-l+3)n+B$	$(s+1)n+B$
13	$j=1,B,\ldots,n$	$\lceil n/B\rceil$	$\hat{\mathbf{v}}$; B Komp. von $\mathbf{w}_{l+1}, \ldots, \mathbf{w}_s, \Delta\eta, \mathbf{e}$; $B+D-1$ Komp. von $\mathbf{w}_l$	$\hat{\mathbf{v}}, \mathbf{w}_l, \ldots, \mathbf{w}_s, \Delta\eta, \mathbf{e}$	$D+(s-l+4)B-1$	$(s-l+3)n+B$
15	$k=0,\ldots,B-1$	B	$\hat{\mathbf{v}}[k+1]$; D Komp. von $\mathbf{w}_l$	$\hat{\mathbf{v}}$; $B+D-1$ Komp. von $\mathbf{w}_l$	$D+1$	$2B+D-1$
16	$i=l+1,\ldots,s$	$s-l$	$\hat{\mathbf{v}}$; B Komp. von $\mathbf{w}_i$	$\hat{\mathbf{v}}$; B Komp. von $\mathbf{w}_{l+1}, \ldots, \mathbf{w}_s$	$2B$	$(s-l+1)B$
17	$k=0,\ldots,B-1$	B	$\hat{\mathbf{v}}[k+1], \mathbf{w}_i[j+k]$	$\hat{\mathbf{v}}$; B Komp. von $\mathbf{w}_i$	2	$2B$
18	$k=0,\ldots,B-1$	B	$\hat{\mathbf{v}}[k+1], \Delta\eta[j+k]$	$\hat{\mathbf{v}}$; B Komp. von $\Delta\eta$	2	$2B$
19	$k=0,\ldots,B-1$	B	$\hat{\mathbf{v}}[k+1], \mathbf{e}[j+k]$	$\hat{\mathbf{v}}$; B Komp. von $\mathbf{e}$	2	$2B$

Tab. 2.3: Arbeitsräume der in den allgemeinen sequentiellen Implementierungen verwendeten Schleifen.

Die vektororientierte Implementierung (A) ist charakterisiert durch die große Zahl von 6 von insgesamt 8 innersten Schleifen mit $|R_I| \leq 3$. Eine Ausnahme bilden nur die beiden innersten Schleifen, in denen die Funktionsauswertung aufgerufen wird, wodurch insgesamt $|R_I| = D + 1$ Vektorelemente referenziert werden. Alle diese Schleifen laufen über die Systemdimension, und dementsprechend gilt $|I| = n$ und $|R_L| = 2n$ bzw. $|R_L| = 3n$. Der Berechnungskernel besteht aus drei Phasen: der Berechnung der Stufenvektoren, der Berechnung der Vektoren $\Delta\eta$ und $\mathbf{e}$ sowie der Berechnung des Skalierungsvektors $\mathbf{s}$. Prägend für das Lokalitätsverhalten der Implementierung sind nur die ersten beiden Phasen, deren Arbeitsraum insgesamt $(s + 3)n$ bzw. $(s + 1)n$ Elemente enthält. Der Arbeitsraum R_I der i-Schleife, mit deren Hilfe der Argumentvektor $\mathbf{w}$ berechnet wird, besteht aus $2n$ Elementen.

Implementierung (E) enthält deutlich weniger Schleifen als Implementierung (A). Die erste Berechnungsphase dieser Implementierung ruft die Funktionsauswertung für den Vektor η zur Berechnung des ersten Stufenvektors auf und berechnet gleichzeitig die Skalierungsfaktoren. Die zugehörige Schleife referenziert pro Iteration $|R_I| = D + 2$ und insgesamt $|R_L| = 3n$ Vektorelemente. Größeren Einfluß auf das Lokalitätsverhalten haben die übrigen zwei Berechnungsphasen zur Bestimmung der verbleibenden Stufenvektoren und der Vektoren $\Delta\eta$ und $\mathbf{e}$. Jede dieser beiden Phasen enthält wie Implementierung (A) eine Schleife mit $|R_I| \leq 3$, die allerdings im Gegensatz zu Implementierung (A) über die Stufen iterieren und maximal $(s - 1)$-mal bzw. $(s - 2)$-mal ausgeführt werden. Diese beiden Schleifen werden jeweils von Schleifen über die Systemdimension umschlossen mit $|R_I| \leq s + 2$. Die Berechnungsphase zur Bestimmung der Stufenvektoren enthält darüber hinaus eine Schleife zur Durchführung der Funktionsauswertungen mit $|R_I| = D + 1$ und $|R_L| = 2n$. Für beide Berechnungsphasen beträgt die jeweilige Gesamtgröße des Arbeitsraums $|R_L| = (s + 2)n$.

Die Anzahl der Schleifen innerhalb der Implementierung (D) ist mit insgesamt 5 noch einmal deutlich geringer als die Zahl der Schleifen in Implementierung (E). Da die Berechnung der Vektoren $\Delta\eta$ und $\mathbf{e}$ mit der Stufenberechnung verschmolzen wurde, enthält diese Implementierung nur zwei Berechnungsphasen: die Berechnung der ersten Stufe zusammen mit der Berechnung der Skalierungsfaktoren sowie die Berechnung der restlichen $s - 1$ Stufen. Die innersten Schleifen beider Phasen besitzen einen Arbeitsraum mit $|R_I| \leq 2$ und $|I| \leq s - 1$. Für die umschließenden Schleifen gilt $|R_I| \leq D + s + 2$ und $|R_L| = (s + 3)n$ bzw. $|R_L| \leq (s + 1)n$. Diese Werte entsprechen auch dem Gesamtarbeitsraum der jeweiligen Berechnungsphase.

Die Implementierung (Dblock) weist konstruktionsbedingt viele charakteristische Merkmale sowohl der Implementierung (D) als auch der Implementierung (A) auf. Sie enthält insgesamt 7 innerste Schleifen mit $|R_I| \leq 3$, $|R_L| \leq 3B$ und $|I| = B$. Für die beiden innersten Schleifen, in denen die Funktionsauswertung aufgerufen wird, gilt $|R_I| = D + 1$, $|R_L| = 2B + D - 1$ und $|I| = B$. Das führt dazu, daß die beiden j-Schleifen, die über die Systemdimension iterieren, einen Arbeitsraum mit $|R_I| \leq D + (s + 4)B - 1$ und $|R_L| \leq (s + 3)n + B$ besitzen.

Arbeitsmengen der Implementierungen

Die Betrachtung der Arbeitsräume allein genügt noch nicht, um das Lokalitätsverhalten der Implementierungen bewerten zu können. Deshalb werden zusätzlich die Arbeitsmengen der Implementierungen betrachtet (vgl. Denning 1968; Woo u. a. 1995).

Definition 2.4 (Arbeitsmenge)**.** *Arbeitsmengen* (englisch: *working sets*) definieren sich durch Cachegrößen, ab denen kritische Teile des Arbeitsraumes eines Programms im Cache gespeichert werden können. Dies entspricht den Stellen, an denen sich der Anstieg der Funktion, die die Anzahl der Cache-Fehlzugriffe in Abhängigkeit von der Cachegröße beschreibt, stark ändert.

Die charakteristischen Arbeitsmengen eines Programms können durch Simulationen von Programmläufen unter Verwendung eines einstufigen Caches mit verschiedenen Größen ermittelt werden. Sie sind abhängig von den Eigenschaften des simulierten Caches (Assoziativität, Zeilengröße, Ersetzungsstrategie) sowie der Zuordnung von Datenelementen zu Cachezeilen. Dieser Ansatz wird in Abschnitt 2.4.3 verfolgt. An dieser Stelle soll zunächst versucht werden, anhand der ermittelten Arbeitsräume im Zusammenhang mit der Programmstruktur auf die resultierenden Arbeitsmengen zu schließen. Dabei gehen wir von einem vollassoziativen Cache aus, denn bei einer geringeren Assoziativität müßten Konfliktfehlzugriffe betrachtet werden, die von den konkreten Adressen der Vektoren abhängig sind.

Am effizientesten arbeiten alle Implementierungen, wenn alle von ihnen verwendeten Datenstrukturen im Cache zwischengespeichert werden können. In diesem Fall treten lediglich zwanghafte Cache-Fehlzugriffe auf, wenn Daten erstmalig referenziert werden. Die Anzahl der Fehlzugriffe entspricht in diesem Fall der Größe des Arbeitsraumes der Implementierung. Ist der Cache zu klein, um alle Daten aufzunehmen, kommt es zusätzlich zu Kapazitätsfehlzugriffen. Im Extremfall, d. h. bei einer Cachegröße von 0 Byte, entspricht die Anzahl der Fehlzugriffe der Anzahl der Speicherzugriffsoperationen. Die Funktion, die die Anzahl der Cache-Fehlzugriffe in Abhängigkeit von der Cachegröße beschreibt, verläuft folglich monoton fallend zwischen diesen beiden Extrempunkten. Wir interessieren uns dafür, an welchen Stellen sich der Anstieg der Funktion ändert.

Die wichtigste Arbeitsmenge der Implementierungen ist der in Tab. 2.2 angegebene Speicherplatzbedarf für die jeweils verwendeten Datenstrukturen. Die Implementierungen (A) und (E) besitzen den gleichen Speicherplatzbedarf von $(s+5)n$. Der Speicherplatzbedarf der Implementierung (D) fällt demgegenüber mit $(s+3)n$ geringer aus, so daß Implementierung (D) diesbezüglich ein besseres Lokalitätsverhalten besitzt. Implementierung (Dblock) besitzt mit $(s+3)n+B$ eine nahezu mit Implementierung (D) identische Arbeitsmenge, da B in der Regel klein gegenüber n gewählt wird.

Implementierung (A). Betrachten wir zunächst Implementierung (A). Charakteristisch für diese Implementierung sind 6 Schleifen über die Systemdimension mit $|R_I| = 2$ bzw. $|R_I| = 3$ und $|R_L| = 2n$ bzw. $|R_L| = 3n$. Alle Schleifen mit $|R_L| = 2n$ werden in der ersten Berechnungsphase nacheinander ausgeführt, wenn man sich die l-Schleife und die i-Schleife aufgerollt vorstellt. Mit Ausnahme der ersten Schleife zur Berechnung der ersten Stufe laufen alle diese Schleifen über den Argumentvektor **w** sowie einen zweiten Vektor, der sich zwischen direkt aufeinanderfolgenden Schleifen in der Regel unterscheidet. $2n$ ist deshalb eine wichtige Arbeitsmenge, da ab einer Cachegröße $2n$ eine Wiederverwendung des Argumentvektors möglich wird. Steigt die Cachegröße auf $3n$, wird die Wiederverwendung der Vektoren $\Delta\eta$ und **e** zwischen den j-Schleifen der zweiten Berechnungsphase möglich. $3n$ tritt deshalb ebenfalls als Arbeitsmenge auf. Jede weitere Steigerung der Cachegröße um n bringt eine zusätzliche Verbesserung, weil hierdurch weitere Vektoren zwischen Iterationen der l-Schleife in Zeile 4 wiederverwendet werden können, die einen Arbeitsraum R_I mit $|R_I| = (l+2)n$ besitzt, wobei l von 2 bis s läuft.

Da die 6 Schleifen mit $|R_I| = 2$ bzw. $|R_I| = 3$ über je zwei bzw. drei Vektoren iterieren, sollten im Cache mindestens zwei bzw. drei Cachezeilen verfügbar sein, um die jeweils referenzierten Blöcke dieser Vektoren aufnehmen zu können, ohne daß sie sich gegenseitig verdrängen. Sobald genügend Cachezeilen verfügbar sind, ermöglicht dies eine Ausnutzung der räumlichen Lokalität bezüglich der Vektoren. Dieser Effekt ist um so größer, je länger die Cachezeilen sind.

Eine weitere Arbeitsmenge wird durch die Funktionsauswertung der rechten Seite des Differentialgleichungssystems bestimmt, die in einer Schleife für jede Komponente aufgerufen wird. Damit diese Schleifen effizient arbeiten können, sollte der Cache mindestens $D+1$ Werte aufnehmen können. Dann wird sowohl eine zeitliche Wiederverwendung von Komponenten des Argumentvektors durch die Funktionsauswertung als auch eine räumliche Wiederverwendung beider beteiligter Vektoren möglich.

Implementierung (E). Die in Implementierung (E) verwendeten Schleifen zur Funktionsauswertung ähneln denen in Implementierung (A), weshalb hier ebenfalls eine Arbeitsmenge von $D+1$ bzw. $D+2$ auftritt. Durch die Umordnung der Schleifen wird die zeitliche Lokalität der Schreibzugriffe auf den Argumentvektor sowie die Vektoren $\Delta\eta$ und **e** jedoch von Anfang an ausgenutzt, so daß es keiner Mindestcachegröße bedarf, um diese zeitliche Wiederverwendung zu aktivieren. Durch die Umordnung der Schleifen können sogar Vektorzugriffe eingespart werden, so daß Implementierung (E) insgesamt weniger Speicherzugriffe benötigt und ein dementsprechend geringeres Potential für Cache-Fehlzugriffe aufweist.

Die j-Schleifen in Zeile 10 und Zeile 19 über die Systemdimension besitzen allerdings einen größeren Arbeitsraum als die j-Schleifen in Implementierung (A). Er beträgt $|R_I| = l+1$ bzw. $|R_I| = s+2$ und $|R_L| = (l+1)n$ bzw. $|R_L| = (s+2)n$ ($l = 2, \ldots, s$). Für eine gute Ausnutzung der räumlichen Lokalität in dieses Schleifen sollten daher wenigsten $l+1$ bzw. $s+2$ Cachezeilen verfügbar sein.

Wie in Implementierung (A) findet auch innerhalb der Implementierung (E) eine Wiederverwendung zwischen Schleifen mit $|R_L| = 2n$ bzw. $|R_L| = 3n$ statt. Diese ist jedoch weniger charakteristisch, da nur wenige derartige Schleifen enthalten sind. Es ist daher zu erwarten, daß Arbeitsmengen, die einem größeren Vielfachen von n entsprechen und so die zeitliche Wiederverwendung der Stufenvektoren zwischen

aufeinanderfolgenden Iterationen der l-Schleife in Zeile 8 mit $|R_I| = (l+2)n$, $l = 2, \ldots, s$, ermöglichen, im Verhältnis stärker ausgeprägt sind.

Implementierung (D). Wie Implementierung (E) verfolgt auch diese Implementierung die Strategie der Optimierung der zeitlichen Lokalität. Hier wird jedoch die Lokalität der Lesezugriffe auf Stufenvektorkomponenten ausgenutzt, wodurch im Vergleich zu Implementierung (A) insgesamt weniger Speicherzugriffe erforderlich sind.

Implementierung (D) enthält zwei Berechnungsphasen: die Berechnung der ersten Stufe zusammen mit der Berechnung der Skalierungsfaktoren und die Berechnung der verbleibenden Stufen. Die Berechnung der Vektoren $\Delta\eta$ und $\mathbf{e}$ wurde mit der Stufenberechnung verschmolzen. Die erste Phase kann bezüglich des Zugriffsverhaltens wie eine erste Iteration der zweiten Phase betrachtet werden. Es wird lediglich ein zusätzlicher Vektor, der Skalierungsvektor $\mathbf{s}$, geschrieben.

Da die i-Schleifen auf der innersten Verschachtelungsebene maximal $(s-1)$-mal ausgeführt wird, ergibt sich eine wichtige Arbeitsmenge dieser Implementierung aus dem Arbeitsraum der j-Schleife mit $|R_I| = D + s - l + 2$ mit $l = 2, \ldots, s$. Erst wenn der Cache diese Arbeitsmenge aufnehmen kann, wird eine effiziente räumliche Wiederverwendung von Vektorelementen möglich. Dies hat zur Folge, daß das Potential dieser Implementierung zur Ausnutzung der räumlichen Lokalität durch das Zugriffsverhalten des Differentialgleichungssystem bestimmt wird. Insbesondere wenn die Zugriffsdistanz des Differentialgleichungssystems sehr groß ist, z. B. wenn alle Komponenten des Argumentvektors von der Funktionsauswertung benutzt werden, und der Cache nicht ausreicht, um die von der Funktionsauswertung zugegriffenen Daten zwischenzuspeichern, werden alle anderen Vektoren aus dem Cache verdrängt, und die Ausnutzung der räumlichen Lokalität ist nur noch innerhalb der Funktionsauswertung möglich.

Eine effiziente zeitliche Wiederverwendung vollständiger Vektoren kann zwischen Iterationen der l-Schleife erfolgen. Dies ist allerdings erst möglich, wenn der Cache den Arbeitsraum R_L der j-Schleife vollständig aufnehmen kann. Daraus ergeben sich Arbeitsmengen von $|R_L| = (s-l+3)n$ mit $l = 2, \ldots, s$. Die kleinste dieser Arbeitsmengen beträgt also $3n$, und jede Vergrößerung des Caches um n Elemente führt zu einer weiteren Verbesserung der zeitlichen Wiederverwendung der Vektoren, bis alle verwendeten Vektoren im Cache gespeichert werden können.

Implementierung (Dblock). Implementierung (Dblock) ist ähnlich aufgebaut wie Implementierung (D). Durch das Tiling der j-Schleifen vergrößern sich jedoch einige Arbeitsmengen und neue entstehen. Die Anzahl der Speicherzugriffe erhöht sich, da die skalare Variable $\tilde{v}$ durch einen Vektor $\hat{\mathbf{v}}$ ersetzt wurde.

Es wurde bereits erwähnt, daß sich der Gesamtspeicherplatz von (Dblock) gegenüber (D) um B Elemente erhöht. Das gleiche trifft auf die Arbeitsmenge zu, die durch den Arbeitsraum der j-Schleife gebildet wird, welcher jetzt $|R_L| = (s-l+3)n + B$ mit $l = 2, \ldots, s$ beträgt. Da B in der Regel klein gegenüber n gewählt wird, kann man davon ausgehen, daß sich beide Implementierung ab einer Cachegröße von $3n$ sehr ähnlich verhalten.

Durch das Tiling sind mehrere Schleifen mit $|R_L| = 2B$ bzw. $|R_L| = 3B$ eingeführt worden, die alle schreibend oder lesend auf den temporären Vektor $\hat{\mathbf{v}}$ zugreifen. Dadurch entstehen zwei Arbeitsmengen der Größe $2B$ bzw. $3B$, deren Speicherung im Cache eine Wiederverwendung des Vektors $\hat{\mathbf{v}}$ ermöglicht. Zwei Tile-Schleifen, die Funktionsauswertungen ausführen, haben einen Arbeitsraum von $|R_L| = 2B + D - 1$. Dieser Arbeitsraum bildet ebenfalls eine Arbeitsmenge, da sich durch einen entsprechend großen Cache die zeitliche Wiederverwendung des Vektors $\hat{\mathbf{v}}$ verbessern würde.

Für die Ausnutzung der räumlichen Lokalität sind mehrere Arbeitsmengen verantwortlich. Da die Tile-Schleifen nur jeweils 2 oder 3 Vektoren referenzieren, genügen für meisten Tile-Schleifen 2 bzw. 3 Cachezeilen für die Ausnutzung der räumlichen Lokalität. Lediglich zwei Tile-Schleifen führen Funktionsauswertungen durch, woraus sich für diese Schleifen zur Ausnutzung der räumlichen Lokalität die ebenfalls in den Implementierungen (A) und (E) auftretende Arbeitsmenge von $D + 1$ ergibt. Für die Nutzung der räumlichen Lokalität zwischen aufeinanderfolgenden Iterationen der j-Schleifen muß der Arbeitsraum R_I dieser Schleifen mit $|R_I| = D + (s-l+4)B$, $l = 2, \ldots, s$, im Cache gehalten werden können.

Zusammenfassung und Vergleich

Zusammenfassend lassen sich folgende Aussagen treffen:

- Die Berechnungskernel der Implementierungen bestehen aus einer verschachtelten Struktur von Schleifen. Jede dieser Schleifen besitzt zwei spezifische Arbeitsräume, R_I und R_L, welche die innerhalb dieser Schleife referenzierten Datenelemente umfassen (Tab. 2.3).
- Innerhalb der Implementierungen treten eine Reihe von Arbeitsmengen auf, welche die Ausnutzung der zeitlichen oder der räumlichen Lokalität ermöglichen. Sie lassen sich aus den Arbeitsräumen im Zusammenhang mit der Programmstruktur herleiten.
- Zeitliche Lokalität:
  1. Die wichtigste Arbeitsmenge einer Implementierungen ist die Gesamtmenge der während eines Zeitschrittes referenzierten Daten. Dies entspricht dem Speicherplatzbedarf der Implementierung (siehe Tab. 2.2). Können alle Daten eines Zeitschrittes im Cache gespeichert werden, treten keine (Kapazitäts-)Fehlzugriffe auf.
     - Der Speicherplatzbedarf von (A) und (E) ist identisch und beträgt $(s+5)n$.
     - Der Speicherplatzbedarf von (D) und (Dblock) ist nahezu identisch und beträgt ca. $(s+3)n$.
     - Infolgedessen genügt für (D) und (Dblock) eine kleinere Cachegröße als für (A) und (E), um die bestmögliche Laufzeit dieser Implementierungen zu erzielen.
  2. Ist die Cachegröße geringer, ist bis zu einer bestimmten Grenze noch eine Wiederverwendung vollständiger Vektoren mit Systemgröße möglich.
     - Für Implementierung (A) und (E) liegt diese Grenze bei $2n$, da mehrere innerste Schleifen über die Systemdimension laufen und dabei 2 Vektoren elementweise referenzieren. Da (E) jedoch nur eine solche Schleife enthält, ist diese Arbeitsmenge hier nicht sehr ausgeprägt. Einen stärkeren Einfluß dürfte hier eine Arbeitsmenge von $3n$ ausüben.
     - Für Implementierung (D) und (Dblock) liegt diese Grenze bei $3n$. Dies entspricht der kleinsten Anzahl von Vektoren, die während eines Durchlaufs der äußeren Schleife über die Stufen benutzt wird.
     - Zwischen dieser Untergrenze und dem Gesamtspeicherplatzbedarf erlaubt die Vergrößerung des Caches um jeweils n Elemente die Wiederverwendung eines zusätzlichen Vektors und führt damit zu einer Senkung der Anzahl der Cache-Fehlzugriffe.
  3. Für Cachegrößen unterhalb der genannten Grenzen ist eine Wiederverwendung von Vektoren nur in Implementierung (Dblock) möglich. Durch das Tiling der Schleifen über die Systemdimension mit einer Blockgröße von B sind Arbeitsmengen von $2B$, $3B$ und $2B+D-1$ entstanden. Ab einer entsprechenden Cachegröße ist eine Wiederverwendung des Vektors $\hat{\mathbf{v}}$ der Größe B möglich.
  4. Für noch kleinere Cachegrößen kommt nur noch eine zeitliche Wiederverwendung einzelner Komponenten in Frage.
     - Aufeinanderfolgende Funktionsauswertungen für benachbarte Komponenten können, abhängig von der konkreten Realisierung des Differentialgleichungssystems, einen Teil der zugegriffenen Komponenten des Argumentvektors wiederverwenden. Dazu müssen, abhängig vom konkreten Zugriffsmuster der Differentialgleichung, in der Regel $D = 2d(\mathbf{f}) + 1$ Komponenten des Argumentvektors sowie, abhängig von der Implementierung, einige zusätzliche Vektorelemente im Cache gespeichert werden können. Für die einzelnen Implementierungen betragen diese Arbeitsmengen:
       - (A): $D+1$,
       - (E): $D+1$ und $D+2$,
       - (D): $D+s+2$ und $D+s-l+2, l=2,\ldots,s$,
       - (Dblock): $D+1$, $D+(s+4)B-1$ und $D+(s-l+4)B-1, l=2,\ldots,s$.
     - Die zeitliche Lokalität bezüglich der Schreibzugriffe auf Argumentvektorkomponenten wird von der Implementierung (E) ausgenutzt. Die erforderlichen Zwischenwerte können in einem Register gespeichert werden, so daß dafür kein Speicherplatz innerhalb des Caches benötigt wird.
     - Analog nutzt Implementierung (D) die zeitliche Lokalität bezüglich der Lesezugriffe auf Stufenvektorkomponenten aus. Dafür wird ebenfalls kein Speicherplatz innerhalb des Caches benötigt.
- Räumliche Lokalität:

1. Die Ausnutzung der räumlichen Lokalität erfolgt durch Zugriffe auf benachbarte Vektorkomponenten innerhalb der Schleifen über die Systemdimension. Infolgedessen muß der Cache groß genug sein, um den Arbeitsraum R_I der betreffenden Schleifen aufnehmen zu können, damit alle während einer Iteration dieser Schleife geladenen Cachezeilen in der nachfolgenden Iteration noch zur Verfügung stehen.
2. Implementierung (A) wurde mit dem Ziel der Ausnutzung der räumlichen Lokalität entworfen. Dementsprechend werden in den Schleifen über die Systemdimension meist nur 2 oder 3 Vektorelemente referenziert. Nur die beiden Schleifen zur Durchführung der Funktionsauswertungen referenzieren $D+1$ Elemente.
3. In Implementierung (E) werden innerhalb der drei Schleifen über die Systemdimension jeweils $D+1$, $s+2$ und $l+1$, $l=2,\ldots,s$, Vektorelemente benutzt.
4. Die beiden Schleifen über die Systemdimension in Implementierung (D) referenzieren jeweils $D+s+2$ bzw. $D+s-l+2$, $l=2,\ldots,s$, Vektorelemente.
5. In Implementierung (Dblock) ist eine gute Ausnutzung der räumlichen Lokalität vor allem innerhalb der Tile-Schleifen gegeben, die nur 2, 3 oder $D+1$ Vektorelemente referenzieren. Die äußeren Schleifen über die Systemdimension mit Inkrement B erfordern dagegen $D+(s+4)B-1$ bzw. $D+(s-l+4)B-1$, $l=2,\ldots,s$, Elemente.
6. Daraus läßt sich schlußfolgern, daß eine Ausnutzung der räumlichen Lokalität im allgemeinen nur innerhalb der Implementierung (A), (E) und (Dblock) möglich ist, da (D) die Speicherung der von der Funktionsauswertung zugegriffenen Datenelemente voraussetzt. Ist jedoch die Zugriffsdistanz des Differentialgleichungssystems hinreichend klein, kann auch Implementierung (D) die räumliche Lokalität effektiv ausnutzen. Für eine optimale Nutzung der räumlichen Lokalität in Implementierung (Dblock) ist eine geeignete Wahl der Blockgröße erforderlich.

- Wie sich die innerhalb der Implementierungen auftretenden Arbeitsmengen auf die Laufzeit auswirken, hängt von vielen Faktoren ab, die im Rahmen einer theoretischen Analyse nur schwer genau erfaßt werden können. Diese Faktoren werden u. a. durch die Zielplattform und den verwendeten Compiler bestimmt.

2.4.3 Simulationsbasierte Analyse des Lokalitätsverhaltens

Um einen detaillierteren Einblick in das Lokalitätsverhalten der Implementierungen, insbesondere die Auswirkungen der Arbeitsmengen auf die Anzahl der Fehlzugriffe zu erhalten, kann man Simulatoren verwenden, die eine Speicherhierarchie mit vorgegebenen Parametern nachbilden und dem Anwender detaillierte Informationen über die ausgeführten Zugriffe liefern. Diesbezüglich gibt es eine Reihe von Ansätzen:

- **Ausführungsgesteuerte Simulation:** Das zu untersuchende, gegebenenfalls modifizierte Binärprogramm wird ausgeführt und während der Ausführung analysiert.
  - **Simulation auf Instruktionsebene:** Der Maschinenbefehlssatz des Zielsystems wird vollständig simuliert. Dadurch ist es möglich, das zu untersuchende Programm ohne Modifikation in binärer Form auf dem simulierten System auszuführen und zu analysieren. Beispiele solcher Simulatoren sind SimpleScalar (Austin u. a. 2002), SimOS (Herrod 1998) und Simics (Magnusson u. a. 2002).
  - **Host-unterstützte Simulation:** Das Binärprogramm wird auf einem Wirtssystem ausgeführt. Beim Eintreten bestimmter Ereignisse, z. B. der Ausführung einer Speicherzugriffsoperation, wird zur Simulationssoftware verzweigt.
- **Protokollgesteuerte Simulation:** Das zu untersuchende, gegebenenfalls modifizierte Binärprogramm wird auf einem Wirtssystem ausgeführt. Während der Ausführung wird ein Protokoll erstellt, das anschließend als Eingabe für den Simulator dient. Ein Cache-Simulator, der auf diese Weise arbeitet, ist Dinero IV (Edler u. Hill).

Für die im folgenden präsentierte Analyse wurde der auf Instruktionsebene arbeitende Simulator Simics (Magnusson u. a. 2002) in der Version 1.6.11 zur Simulation eines SPARC-V9-Befehlssatzes verwendet.

Dieser Simulator wurde bereits erfolgreich von Wallin u. a. (2003) zu Analyse des Lokalitätsverhaltens dreier Berechnungskernel von Lösungsverfahren für partielle Differentialgleichungssysteme eingesetzt. Durch die Simulation auf Instruktionsebene ist es möglich, die Implementierungsvarianten zu untersuchen, ohne den Programmcode verändern zu müssen. Dadurch kann das Lokalitätsverhalten des Programms genauer erfaßt werden, da modifizierter Programmcode zu einem veränderten Lokalitätsverhalten führen kann.

Durchführung der Simulationsexperimente

Um die Arbeitsmengen der Implementierungen identifizieren zu können und um einen Vergleich der Implementierungen anhand der Anzahl der Cache-Fehlzugriffe zu ermöglichen, simulieren wir einen einstufigen Cache und verändern dessen Parameter, d. h. Zeilengröße, Anzahl der Zeilen und Assoziativität. Dabei genügt es, nur eine kleine Anzahl von Zeitschritten zu betrachten, da in jedem Zeitschritt nahezu identischer Code ausgeführt wird. Lediglich zwischen akzeptierten und verworfenen Zeitschritten gibt es Unterschiede im Lokalitätsverhalten, die jedoch unabhängig vom verwendeten Berechnungskernel sind. Die Ausführung einer großen Anzahl von Zeitschritten für relevante Problemgrößen wäre aus Zeitgründen nicht durchführbar, da die Simulationen sehr rechenzeitaufwendig sind und man deshalb zum Ausgleich die Anzahl der Simulationsexperimente reduzieren müßte.

Wir führen darum insgesamt nur 5 Zeitschritte aus. Die ersten beiden Zeitschritte dienen dazu, den Cache aufzuwärmen, d. h. alle Datenelemente mindestens einmal zu referenzieren und in den Cache zu laden. Die Anzahl der Cache-Fehlzugriffe während der letzten drei Zeitschritte wird für die Analyse benutzt. Da innerhalb des Simulators neben dem zu untersuchenden Binärprogramm auch ein Betriebssystem ausgeführt wird (hier: Solaris 9), hilft die zusätzliche Anzahl von Zeitschritten, eventuelle Störungen durch Unterbrechungen durch das Betriebssystem auszugleichen.

Als Testprobleme betrachten wir BRUSS2D-MIX und MEDAKZO. Beide Probleme wurden aus partiellen Differentialgleichungen mit Hilfe der Linienmethode abgeleitet. Eine detaillierte Beschreibung wird in Anhang B gegeben. Für das Verständnis der Simulationsergebnisse ist lediglich die Kenntnis des Zugriffsverhaltens der Funktionsauswertung notwendig.

Im Fall von BRUSS2D-MIX greift die Komponentenfunktion $f_j(t, \mathbf{w})$ auf die Komponenten des Argumentvektors $\mathbf{w}$ mit Index

$$j-2N,\ j-2,\ j,\ j+1,\ j+2 \text{ und } j+2N$$

zu, falls j ungerade ist und diese Indizes auf existierende Komponenten verweisen, sie also einen Wert zwischen 1 und n besitzen. Ist j gerade, werden die Komponenten mit Index

$$j-2N,\ j-2,\ j-1,\ j,\ j+2 \text{ und } j+2N$$

des Argumentvektors benutzt, falls diese verfügbar sind. Die Zugriffsdistanz $d(\mathbf{f})$ beträgt dementsprechend $2N$. N ist ein Parameter, der die Anzahl der Gitterlinien vorgibt, die für die Ortsdiskretisierung des ursprünglichen zweidimensionalen partiellen Differentialgleichungssystems verwendet werden. Die Größe des resultierenden gewöhnlichen Differentialgleichungssystems beträgt $n = 2N^2$, da zwei abhängige Variablen vorhanden sind. In den Simulationsexperimenten mit BRUSS2D-MIX wird das 7-stufige Verfahren DOPRI5(4) verwendet. Die in Implementierung (Dblock) verwendete Blockgröße wurde auf $B = d(\mathbf{f}) = 2N$ gesetzt.

MEDAKZO besitzt eine noch geringere Zugriffsdistanz von $d(\mathbf{f}) = 2$, da für j ungerade nur die Komponenten mit Index

$$j-2,\ j,\ j+1 \text{ und } j+2$$

benutzt werden, falls diese Indizes gültig sind, und für j gerade nur die Komponenten

$$j-1 \text{ und } j,$$

ebenfalls vorausgesetzt, daß diese Indizes gültige Vektorelemente referenzieren. Aufgrund dieser geringen Zugriffsdistanz läßt sich das durch die Programmstruktur der Implementierungen hervorgerufene Lokalitätsverhalten untersuchen, da das Differentialgleichungssystem keine signifikante Arbeitsmenge generiert und deshalb nur einen geringen Einfluß auf das beobachtete Lokalitätsverhalten ausübt. Die Systemgröße

wird auch im Fall von MEDAKZO über einen Parameter namens N gesteuert, wobei sich die Systemgröße zu $n = 2N$ ergibt. Als Lösungsverfahren wird für die Untersuchung des Lokalitätsverhaltens dieses Testproblem das 13-stufige Verfahren DOPRI8(7) benutzt. Da die Zugriffsdistanz hier konstant ist, wurde Implementierung (Dblock) für dieses Testproblem mit einer konstanten Blockgröße von $B = 128$ ausgeführt.

Auswirkung der Assoziativität

Als erstes betrachten wir, wie sich die Assoziativität des Caches auf die Anzahl der Cache-Fehlzugriffe auswirkt. Dazu messen wir für verschiedene Assoziativitäten die Anzahl der Cache-Fehlzugriffe in Abhängigkeit von der Cachegröße unter Verwendung einer konstanten Zeilengröße. Für die nachfolgende Diskussion beziehen wir uns beispielhaft auf eine Zeilengröße von 128 Byte.

Eine hohe Assoziativität dient prinzipiell dazu, die Anzahl der Konfliktfehlzugriffe zu reduzieren, die dadurch entstehen, daß im Wechsel auf mehrere Datenelemente zugegriffen wird, die aufgrund ihrer Speicheradressen in die gleiche Cachemenge fallen und deren Anzahl die Größe der Cachemengen übersteigt. Bei Verwendung eines vollassoziativen Caches treten nach Definition keine Konfliktfehlzugriffe mehr auf.

Da die Speicheradressen der Datenstrukturen deren Abbildung auf die Cachemengen bestimmen und somit wesentlichen Einfluß auf die Anzahl der Konfliktfehlzugriffe ausüben, ist es für das Verständnis der Simulationsergebnisse notwendig, den Aufbau der Datenstrukturen innerhalb des Speichers zu kennen. Die im folgenden betrachteten Implementierungen realisieren die jeweils verwendeten Datenstrukturen wie folgt:

- Speicher für eindimensionale Felder wird für jedes Feld einzeln durch einen entsprechenden Aufruf von `malloc()` angefordert.
- Zweidimensionale Felder mit a Zeilen und b Spalten werden als Block der Größe $a \times b$ allokiert. Zusätzlich wird mittels eines separaten Aufrufs von `malloc()` ein Zeigerfeld der Größe a allokiert, dessen Einträge auf die Anfangspositionen der Zeilen innerhalb des $a \times b$-Blocks gesetzt werden, um Zugriffe der Form $A[i][j]$ zu ermöglichen.
- Für die Speicherung der Verfahrenskoeffizienten werden ein $s \times s$-Feld sowie drei eindimensionale Felder der Größe s verwendet.
- Werden mehrere Stufen- oder Argumentvektoren benutzt, werden diese als zweidimensionales $s \times n$-Feld allokiert. Kommt nur ein einzelner Stufen- oder Argumentvektor vor, wird dieser als eindimensionales Feld realisiert.
- Alle übrigen Vektoren (η, $\Delta\eta$, $\mathbf{e}$, $\mathbf{s}$) werden als eindimensionale Felder gespeichert.

Abbildung 2.10 zeigt Simulationsergebnisse für das Testproblem MEDAKZO mit $N = 15\,000$ bei Verwendung eines Caches mit einer Assoziativität von 1, 2, 4, 8 und 16. Wie erwartet verringert sich die Anzahl der Fehlzugriffe mit zunehmender Assoziativität. Dabei wirkt sich die Änderung der Assoziativität auf die einzelnen Implementierungen in etwa im gleichen Maße aus. Ab einer Assoziativität von 4 sind nur noch geringe Verbesserungen zu beobachten. Mit zunehmender Cachegröße nähern sich die Kurven für die verschiedenen Assoziativitäten einander an. Ein ähnliches Bild ergibt sich für BRUSS2D-MIX mit $N = 250$ (Abb. 2.11). Die Auswirkungen einer geringen Assoziativität sind hier allerdings deutlich geringer, da mit einem 7-stufigen Verfahren anstelle eines 13-stufigen Verfahrens gerechnet wurde, wodurch es weniger häufig zu Interferenzen zwischen Vektoren kommt.

Die möglichen Auswirkungen einer ungünstigen Ausrichtung der Datenstrukturen zeigt Abb. 2.12 für das Testproblem BRUSS2D-MIX. Hier wurde als Vorgabe für die Problemgröße der Parameter $N = 256$ benutzt, woraus eine Vektorgröße von genau 1 MB resultiert. Da es sich bei den untersuchten Cachegrößen – wie auch bei den meisten Cachegrößen realer Computersysteme – um Zweierpotenzen handelt, die folglich Teiler oder Vielfache von 1 MB sind, werden bis zur untersuchten Cachegröße von 4 MB aufgrund der Allokierung als Block die Komponenten mehrerer Stufen- bzw. Argumentvektoren mit gleichem Index auf gleiche Cachemengen abgebildet.[3] Dies wirkt sich besonders drastisch auf die Implementierungen (D)

[3] Da DOPRI5(4) 7 Stufen besitzt, kann ab einer Cachegröße von 8 MB der gesamte 7 MB große Block ohne Überschneidungen mit sich selbst im Cache gespeichert werden.

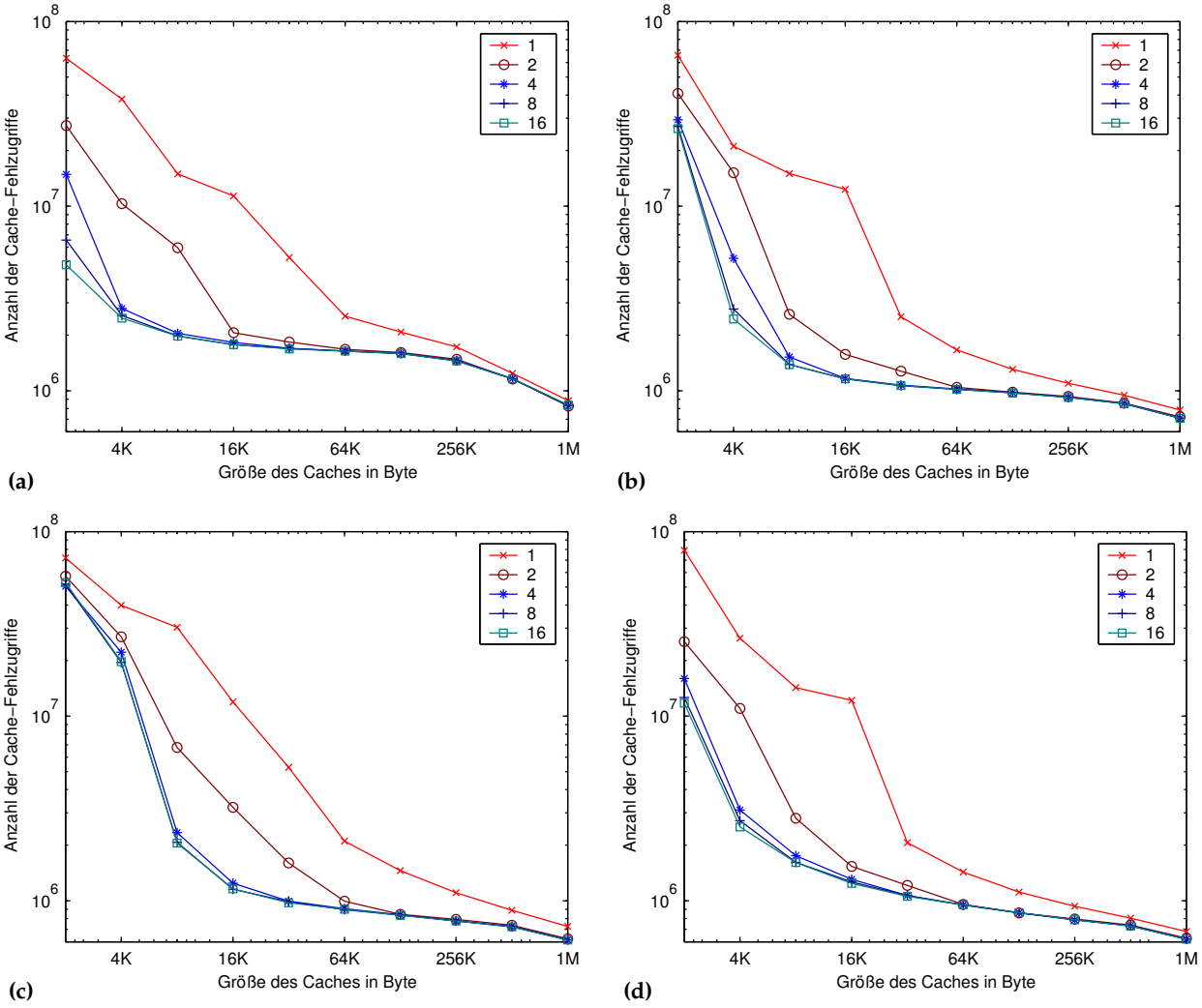


Abb. 2.10: Auswirkung der Assoziativität auf die Anzahl der Cache-Fehlzugriffe für MEDAKZO, $N = 15\,000$ bei einer Zeilengröße von 128 Byte. **(a)** Implementierung (A), **(b)** Implementierung (E), **(c)** Implementierung (D), **(d)** Implementierung (Dblock).

und (E) aus, da sie in ihren innersten Schleifen Komponenten mehrerer dieser Vektoren mit gleichem Index benutzen. Ist die Assoziativität, also die Größe der Cachemengen nicht ausreichend, um die Elemente aller innerhalb der innersten Schleife zugegriffenen Vektoren aufzunehmen, kommt es zu einer großen Anzahl von Konfliktfehlzugriffen. Man spricht in diesem Fall von *Thrashing*.

Ungewöhnlich ist in diesem Zusammenhang, daß bei dieser ungünstigen Ausrichtung der Vektoren insbesondere für Implementierung (A) mit 16-facher Assoziativität oft mehr Cache-Fehlzugriffe auftraten als bei Verwendung eines 8- oder 4-fach assoziativen Caches. Die Ursache dafür steht vermutlich im Zusammenhang mit der speziellen Ersetzungsstrategie, die das für die Simulation benutzte Cachemodell verwendet. Hierbei handelt es sich um eine Zufallsstrategie, bei der die zu ersetzende Cachezeile zufällig mit Hilfe eines einfachen Pseudo-Zufallszahlengenerators ausgewählt wird. Bei einer Untersuchung des Einflusses verschiedener Zufallszahlengeneratoren auf die Anzahl der Fehlzugriffe des Berechnungskernels mit Hilfe eines alternativen Cache-Simulators konnte dieser Effekt jedoch bisher nicht reproduziert werden.

Um das Problem einer ungünstigen Ausrichtung der Vektoren im Speicher zu verdeutlichen, zeigt Abb. 2.13 für Implementierung (D) zwei mögliche Fälle, wie die beteiligten Datenstrukturen auf Cachemengen abgebildet werden könnten. Aus der Abbildung ist ebenfalls die Anzahl der Zugriffe auf die zugehörigen Speicherzellen ersichtlich. Für diese Darstellung wurde ein direkt abbildender Cache der Größe

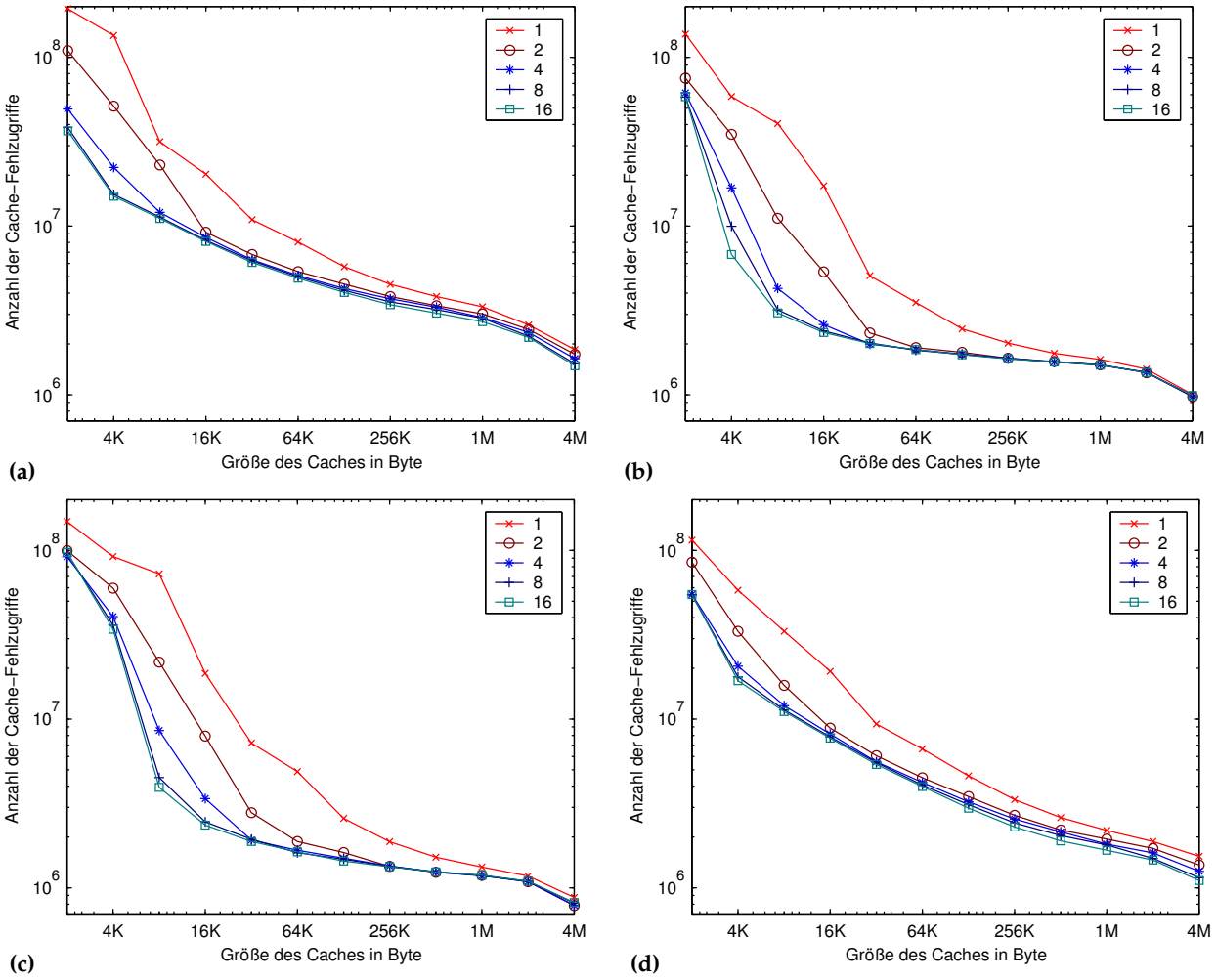


Abb. 2.11: Auswirkung der Assoziativität auf die Anzahl der Cache-Fehlzugriffe für BRUSS2D-MIX, $N = 250$ bei einer Zeilengröße von 128 Byte. **(a)** Implementierung (A), **(b)** Implementierung (E), **(c)** Implementierung (D), **(d)** Implementierung (Dblock).

32 KB mit einer Zeilengröße von 32 Byte zugrunde gelegt. Als Testproblem wurde BRUSS2D-MIX verwendet, und die Problemgröße wurde so gewählt, daß die Größe eines Vektors genau der halben Cachegröße, also 16 KB entspricht. Zur Ermittlung der in der Grafik dargestellten Informationen wurden die im Programmcode des Berechnungskernels der Implementierung verwendeten Standarddatentypen durch protokollierende Datentypen ersetzt, die ein Protokoll der ausgeführten Speicherzugriffe sowie der zugehörigen Variablennamen generieren.

Für den ersten, in Abb. 2.13 (a) dargestellten Fall wurden alle Vektoren, einschließlich der Argumentvektoren, mit separaten, direkt aufeinanderfolgenden Aufrufen von `malloc()` allokiert. Anhand der Abbildung kann man erkennen, daß die Vektoren infolgedessen relativ dicht benachbart im Speicher liegen. Zwischen zwei aufeinanderfolgenden Vektoren befindet sich allerdings ein Zwischenraum von einigen Bytes, der u. a. daraus resultiert, daß `malloc()` an dieser Stelle Informationen ablegt, die später für die Freigabe des Speicherblocks benötigt werden. Als Konsequenz daraus ergibt sich, daß sich die Adressen der Cachemengen für gleiche Komponenten aufeinanderfolgend allokierter Vektoren jeweils um diesen Betrag unterscheiden. Für den zweiten Fall (Abb. 2.13 (b)) wurden die Anfangsadressen aller Vektoren durch Padding so ausgerichtet, daß sie auf den Anfang des Caches abgebildet werden. Dadurch werden alle Vektorkomponenten mit gleichem Index auf die gleiche Cachemenge abgebildet. Dies entspricht dem in Abb. 2.12 vorliegenden Fall. Die beiden dargestellten Fälle wurden anhand der erzeugten Speicherzugriffsprotokol-

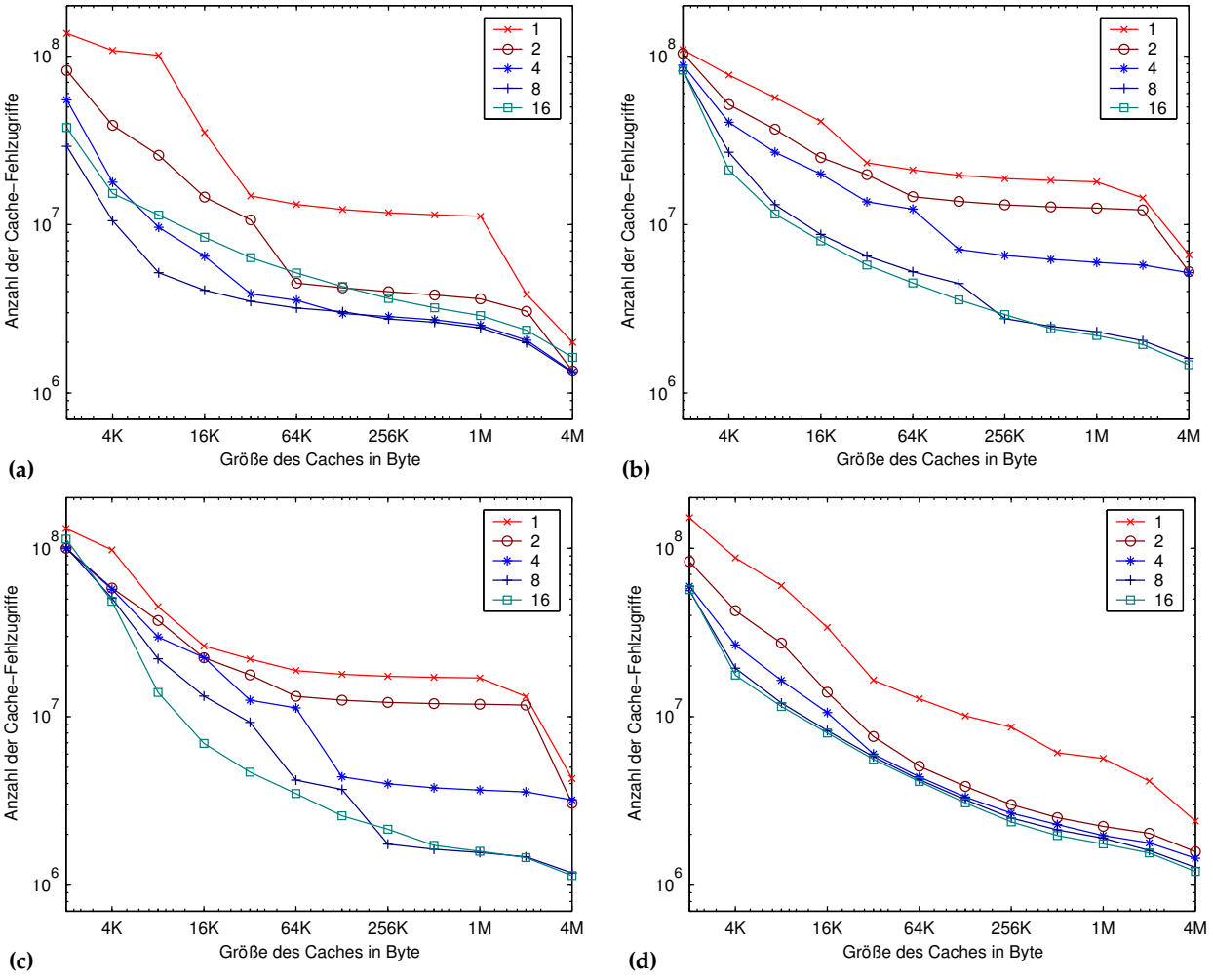


Abb. 2.12: Auswirkung der Assoziativität auf die Anzahl der Cache-Fehlzugriffe für BRUSS2D-MIX, $N = 256$ bei einer Zeilengröße von 128 Byte. **(a)** Implementierung (A), **(b)** Implementierung (E), **(c)** Implementierung (D), **(d)** Implementierung (Dblock).

le mit Hilfe eines speziellen Cache-Simulators simuliert. Die Ergebnisse der beiden Simulationen sind in Abb. 2.14 dargestellt. Das erwartete Ansteigen der Anzahl der Fehlzugriffe durch die gezielt ungünstige Ausrichtung der Vektoren ist hier deutlich erkennbar.

Abbildung 2.13 (a) zeigt auch, warum ein vierfach mengenassoziativer Cache in der Regel ausreicht, um eine effiziente Ausführung der Implementierungen zu gewährleisten. Der Grund ist, daß neben den Stufen-, Argument-, Approximations- und Hilfsvektoren genau zwei, in der Regel zusammenhängende Speicherbereiche im Cache gehalten werden müssen. Dabei handelt es sich um den Aufrufstack, der sich in fast allen Fällen an einer festen Adresse befindet, und die dynamisch allokierten Verfahrenskoeffizienten. Aus 2.13 ist ersichtlich, daß die auf dem Stack befindlichen Daten und die Verfahrenskoeffizienten wesentlich häufiger innerhalb des Programms benutzt werden als die einzelnen Vektorkomponenten. Entsprechend wichtig ist es, daß diese beiden Speicherbereiche ständig im Cache gehalten werden können, ohne daß sie durch Interferenzen mit Vektoren verdrängt werden können. Sind die Adressen der Vektoren geeignet ausgerichtet, um Interferenzen zwischen Vektoren zu minimieren, genügt ein vierfach assoziativer Cache, um den Fall abzusichern, daß sich Aufrufstack und Verfahrenskoeffizienten im Cache überlagern. Selbst bei optimaler Ausrichtung aller Datenstrukturen ist mindestens noch ein zweifach assoziativer Cache erforderlich, da sich aufgrund der Größe der Vektoren eine Überlagerung von Vektoren mit den Speicherbereichen für den Stack und die Verfahrenskoeffizienten in der Regel nicht vermeiden läßt. Eine solche

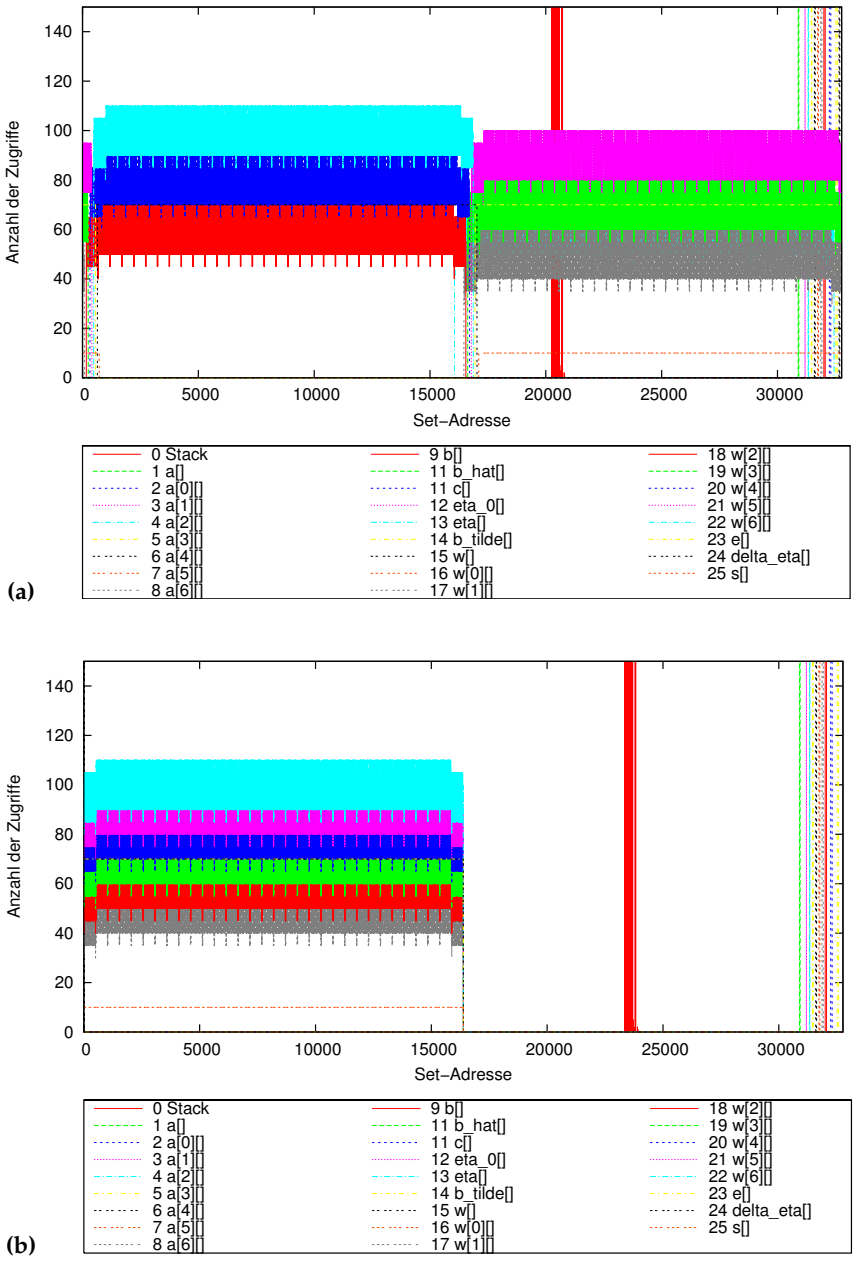
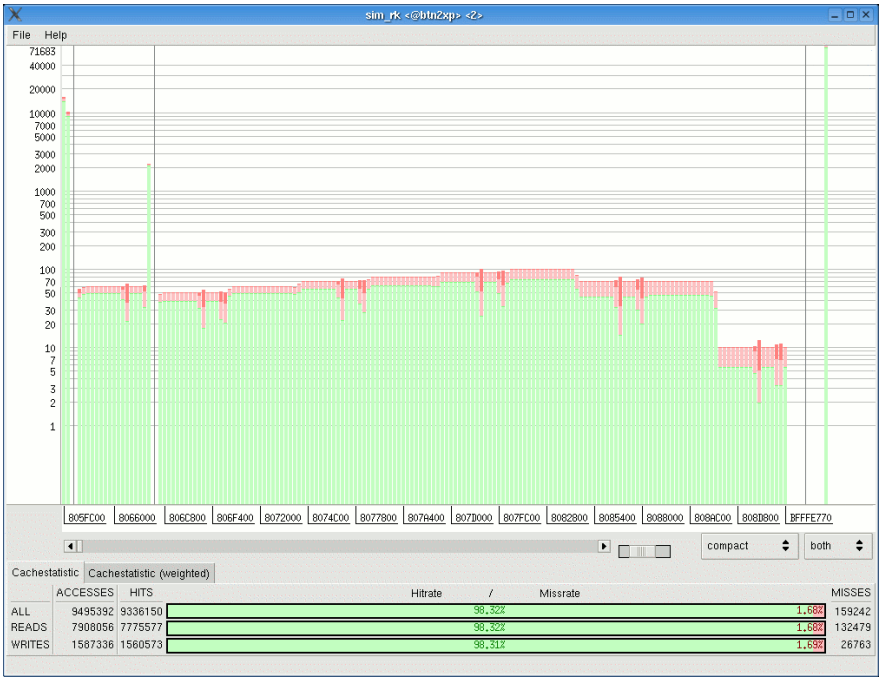
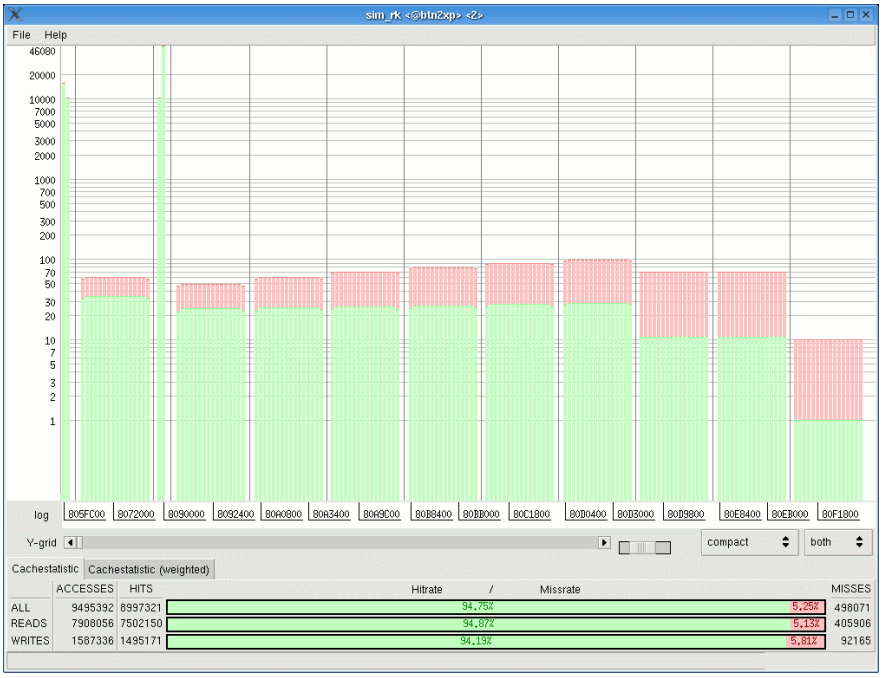


Abb. 2.13: Mögliche Abbildung der in Implementierung (D) verwendeten Datenstrukturen auf Mengen eines mengenassoziativen Caches mit 32 768 Mengen. **(a)** Verwendung der von der Speicherverwaltung des Betriebssystems zurückgegebenen Adressen, **(b)** Ausrichtung der Argument- und Hilfsvektoren, so daß die Anfangsandressen dieser Vektoren auf die Cachemenge mit der Adresse 0 abgebildet werden. Zusätzlich ist auf der Ordinatenachse für jedes Datenelement die Anzahl der Zugriffe ersichtlich. Der Wertebereich wurde zur Verbesserung der Darstellung der Zugriffe auf die Argument- und Hilfsvektoren auf das Intervall $[0; 150]$ eingeschränkt.



(a)



(b)

Abb. 2.14: Ergebnisse der Cache-Simulation eines Programmlaufes der Implementierung (D) auf einem direkt abbildenden Cache der Größe 32 KB mit einer Zeilengröße von 32 Bytes. Abgebildet ist die Anzahl der Cache-Treffer (grün) und die Anzahl der Cache-Fehlzugriffe (rot) für jeweils vier Cachezeilen. **(a)** Verwendung der von der Speicherverwaltung des Betriebssystems zurückgegebenen Adressen, **(b)** Ausrichtung der Argument- und Hilfsvektoren, so daß alle diese Vektoren auf die Menge mit der Adresse 0 abgebildet werden.

optimale Anordnung setzt voraus, daß der Cache groß genug ist, um Stack und Verfahrenskoeffizienten vollständig aufzunehmen und eine hinreichend große Spreizung der Vektoradressen zu ermöglichen.

Auswirkung der Zeilengröße und Identifikation der Arbeitsmengen

Um die auftretenden Arbeitsmengen und den Einfluß der Zeilengröße zu untersuchen, sollte möglichst ein vollassoziativer Cache verwendet werden, um Konfliktfehlzugriffe aufgrund einer zu niedrigen Assoziativität zu vermeiden. Für die folgenden Betrachtungen benutzen wir einen 16-fach mengenassoziativen Cache, da aus den vorausgehenden Untersuchungen bekannt ist, daß diese Assoziativität für die zu untersuchenden Implementierungen auch unter ungünstigen Bedingungen ausreichend ist. Außerdem stellt eine beschränkte Assoziativität einen besseren Praxisbezug her, da in aktuellen Computersystemen vollassoziative Caches nicht gebräuchlich sind.

In den Abb. 2.15 und 2.16 ist die Anzahl der Cache-Fehlzugriffe für verschiedene Zeilengrößen zwischen 8 Byte und 1024 Byte für die beiden Testprobleme MEDAKZO ($N = 15\,000$) und BRUSS2D-MIX ($N = 250$) dargestellt. Es ist deutlich erkennbar, daß sich die Fehlzugriffskurven bei Veränderung der Zeilengröße ebenfalls stark verändern. Prinzipiell verringert sich für alle Implementierungen die Anzahl der Cache-Fehlzugriffe, wenn die Zeilengröße erhöht wird. Dies liegt daran, daß alle Implementierungen durch die elementweise Iteration über Vektoren eine hohe räumliche Lokalität besitzen. Selbst wenn keine zeitliche Wiederverwendung von Vektorkomponenten stattfindet, entstehen bei einer elementweisen Iteration über einen Vektor jeweils nur (Vektorgröße in Byte)/(Zeilengröße in Byte) Fehlzugriffe. Für die beiden Testprobleme MEDAKZO und BRUSS2D-MIX mit einer Vektorgröße von 235 KB bzw. 977 KB ergeben sich beispielsweise bei einer Zeilengröße von 1024 Byte für eine solche elementweise Iteration über einen Vektor lediglich 235 bzw. 977 Fehlzugriffe, während bei einer heute in realen System typischen Zeilengröße von 64 Byte die Anzahl der Fehlzugriffe 16 mal größer wäre. Voraussetzung dafür ist, daß der Cache ausreichend groß ist, um für jeden an der Iteration beteiligten Vektor mindestens eine Cachezeile zur Verfügung stellen zu können. Da Implementierung (A) eine geringere zeitliche Lokalität aufweist als die übrigen Implementierungen, profitiert sie besonders stark von einer Erhöhung der Zeilengröße. Dies wird vor allem in Abb. 2.16 (a) deutlich, da hier die Kurve mit zunehmender Zeilengröße stärker abfällt.

Die Erhöhung der Zeilengröße hat aber auch negative Auswirkungen. Wie man aus den Abb. 2.15 und 2.16 ersehen kann, vergrößern sich durch die Erhöhung der Zeilengröße die Arbeitsmengen der Implementierungen. Das bedeutet, daß Stellen, an denen ein starkes Absinken der Anzahl der Fehlzugriffe beobachtet werden kann, hin zu höheren Cachegrößen verschoben werden. Der Grund dafür ist die aus der höheren Zeilengröße resultierende schlechtere Platzausnutzung innerhalb des Caches. Infolgedessen steigt die Wahrscheinlichkeit für Konfliktfehlzugriffe, da durch eine höhere Zeilengröße bei gleicher Gesamtgröße des Caches auch die Anzahl der Mengen reduziert wird und folglich eine größere Anzahl von Speicheradressen auf die selbe Cachemenge abgebildet wird. Da sich auch die Gesamtzahl der Zeilen verringert, steigt ebenfalls die Wahrscheinlichkeit von Kapazitätsfehlzugriffen. Die Implementierungen verarbeiten jedoch größtenteils Vektordaten, die zusammenhängend im Speicher abgelegt sind, so daß dies hauptsächlich für Caches mit einer sehr kleinen Anzahl von Cachezeilen (z. B. weniger als 32 Zeilen) ein Problem darstellt.

Ob sich eine Erhöhung der Zeilengröße positiv oder negativ auf die Laufzeit auswirkt, hängt auch davon ab, wie sich die Zeitparameter des Caches durch die Realisierung einer höheren Zeilengröße verändern. Geht man davon aus, daß die Bandbreite des Datenbusses unverändert bleibt, vergrößert sich die für die Übertragung einer Zeile aus dem Hauptspeicher notwendige Zeit, da eine größere Datenmenge transportiert werden muß. Demgegenüber steht, daß weniger Blöcke übertragen werden müssen und demzufolge Latenzzeiten für Adressierung und Auffinden der Daten im Hauptspeicher eingespart werden können.

Die Arbeitsmengen, die sich anhand der Simulationsergebnisse identifizieren lassen, stimmen im wesentlichen mit den Vorhersagen der theoretischen Analyse überein. Allerdings sind nur einige der theoretisch bestimmten Arbeitsmengen ausschlaggebend für den visuellen Verlauf der Kurven. Besonders signifikant sind bei allen Implementierungen die Arbeitsmengen, die durch die zeitliche Wiederverwendung von Vektoren mit Systemgröße entstehen. So sinkt die Zahl der Fehlzugriffe bei allen Implementierungen stark ab, sobald eine bestimmte Mindestanzahl von Vektoren im Cache gespeichert werden kann (2 Vektoren für die Implementierungen (A) und (E) und 3 Vektoren für die Implementierungen (D) und (Dblock), wobei ein Vektor 235 KB für MEDAKZO bzw. 977 KB für BRUSS2D-MIX benötigt). Mit jeder weiteren Vergrößerung

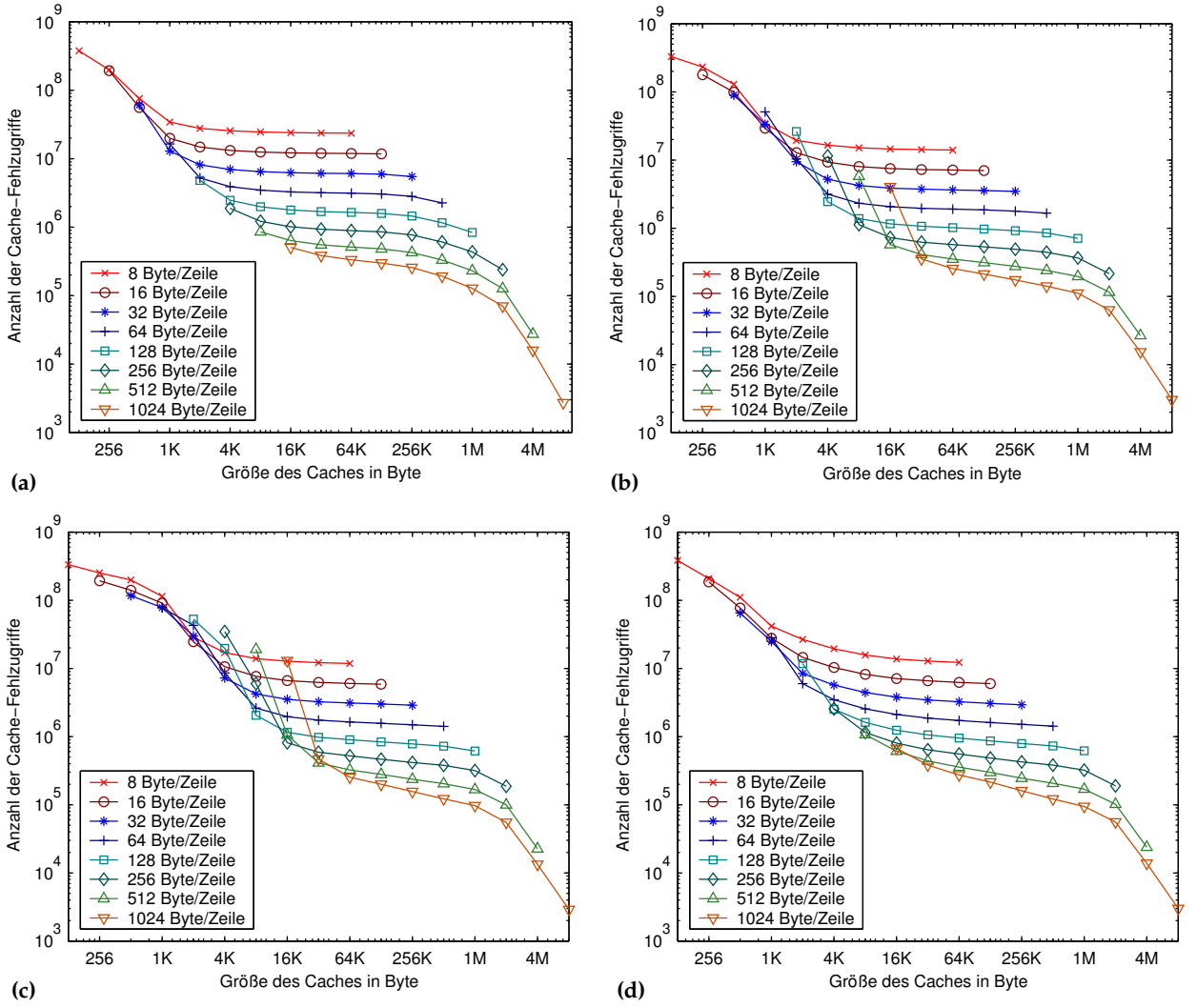


Abb. 2.15: Auswirkung der Zeilengröße auf die Anzahl der Cache-Fehlzugriffe für MEDAKZO, $N = 15\,000$ bei Verwendung eines 16-fach mengenassoziativen Caches. **(a)** Implementierung (A), **(b)** Implementierung (E), **(c)** Implementierung (D), **(d)** Implementierung (Dblock).

des Caches um mindestens eine Vektorgröße sinkt die Anzahl der Fehlzugriffe immer stärker ab, bis alle verwendeten Datenstrukturen im Cache gespeichert werden können. Für MEDAKZO ist dies ab 4220 MB (Implementierung (A) und (E)) bzw. 3750 MB (Implementierung (D) und (Dblock)) der Fall, für BRUSS2D-MIX ab 11 720 MB für die Implementierungen (A) und (E) und ab 9770 MB für die Implementierungen (D) und (Dblock).

Für kleine Cachegrößen zeigen die Simulationsergebnisse, daß für die Implementierungen unterschiedlich große Mindestgrößen des Caches erforderlich sind, bis die Implementierungen effizient arbeiten und sich die Fehlzugriffszahl stabilisiert. Dies liegt zum einen daran, daß ein deutliches Absinken der Fehlzugriffszahl erst dann möglich ist, wenn alle Verfahrenskoeffizienten und der Stackbereich im Cache gehalten werden können. Das Verfahren DOPRI8(7) mit $s = 13$, das zusammen mit dem Testproblem MEDAKZO verwendet wurde, benötigt zur Speicherung aller Koeffizienten 936 Byte. Für das zusammen mit BRUSS2D-MIX verwendete Verfahren DOPRI5(4) mit $s = 7$ werden 336 Byte benötigt. Zusammen mit dem Stackbereich wird demzufolge für alle Implementierungen für die Simulationen mit MEDAKZO mindestens ein 2 KB großer Cache und für die Simulationen mit BRUSS2D-MIX mindestens ein 1 KB großer Cache benötigt, damit diese Voraussetzung für eine effiziente Arbeitsweise der Implementierungen erfüllt ist. Man erkennt

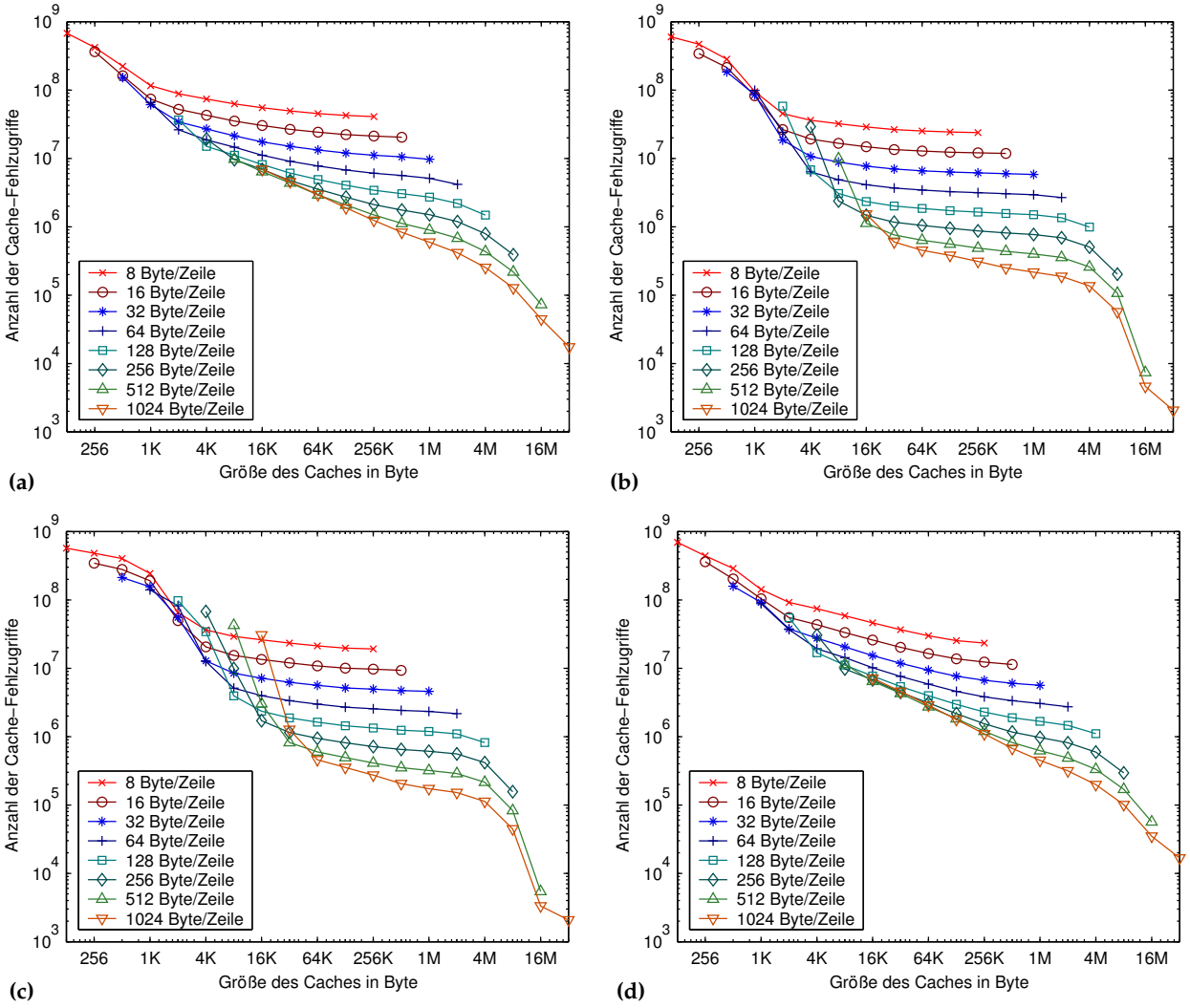


Abb. 2.16: Auswirkung der Zeilengröße auf die Anzahl der Cache-Fehlzugriffe für BRUSS2D-MIX, $N = 250$ bei Verwendung eines 16-fach mengenassoziativen Caches. **(a)** Implementierung (A), **(b)** Implementierung (E), **(c)** Implementierung (D), **(d)** Implementierung (Dblock).

dies in den Abbildungen daran, daß sich die Zahl der Fehlzugriffe für eine Zeilengröße von 8 Byte ab der jeweiligen Cachegröße stabilisiert oder zumindest stark zu fallen beginnt.

Für größere Cachezeilen erfolgt diese Stabilisierung erst ab höheren Cachegrößen. Dies liegt daran, daß für eine effiziente Arbeitsweise eine bestimmte Mindestanzahl von Cachezeilen vorhanden sein muß, so daß im Idealfall für alle Datenstrukturen, die während eines Schleifendurchlaufs gleichzeitig verwendet werden, wenigstens eine Cachezeile zur Verfügung steht. Wieviele Cachezeilen benötigt werden, ist abhängig von der Schleifenstruktur der jeweiligen Implementierung. Am besten kommt Implementierung (A) mit einer kleinen Anzahl von Cachezeilen zurecht, da die dort verwendeten innersten Schleifen nur zwei oder drei Vektoren und gegebenfalls ein oder zwei Verfahrenskoeffizienten referenzieren. Auch für Implementierung (Dblock) kann dank der Tile-Schleifen bereits für kleine Cachegrößen ein stetiges Absinken der Fehlzugriffszahl beobachtet werden. Für die Implementierungen (E) und (D) setzt dieses Absinken erst später ein, ist dafür jedoch deutlich stärker.

Im Vergleich der Simulationen für die beiden unterschiedlichen Testprobleme ist ersichtlich, daß die Zugriffsdistanz des Differentialgleichungssystems charakteristisch auf den Kurvenverlauf einwirkt. Für MEDAKZO ist die Zugriffsdistanz mit $d(\mathbf{f}) = 2$ nahezu vernachlässigbar klein, da die zugegriffenen Kom-

ponenten des Argumentvektors mit zunehmender Zeilengröße sehr oft innerhalb der gleichen Cachezeile liegen. Für BRUSS2D-MIX beträgt die Zugriffsdistanz $d(\mathbf{f}) = 2N = 500$, woraus sich $D = 2d(\mathbf{f}) + 1 = 1001$ ergibt, was einem Speicherbedarf von 8 KB entspricht. Aus diesem Grund ähneln sich die Abb. 2.15 (a)–(d) für MEDAKZO untereinander stärker als die Abb. 2.16 (a)–(d) für BRUSS2D-MIX. Im Vergleich zwischen den beiden Abb. 2.15 und 2.16 kann man feststellen, daß die dargestellten Kurven für die Implementierungen (D) und (E) für beide Testprobleme eine sehr ähnliche Charakteristik besitzen. Für die Implementierungen (A) und (Dblock) unterscheiden sich die Kurvenverläufe für die beiden unterschiedlichen Testprobleme jedoch sehr deutlich. Bei Verwendung von BRUSS2D-MIX fallen die Kurven für (A) und (Dblock) für kleine Cachegrößen weniger steil ab als bei Verwendung von MEDAKZO. Für große Cachegrößen sinkt die Zahl der Fehlzugriffe für BRUSS2D-MIX – ausgehend von einem höheren Niveau – dafür schneller ab als bei Verwendung von MEDAKZO. Besonders stark tritt dieser Effekt für große Zeilengrößen auf. Ursache dafür ist, daß beide Implementierungen eine Arbeitsmenge der Größe $D + 1$ besitzen. Unterschreitet der Cache diese Größe, arbeiten die Schleifen, in denen die Funktionsauswertung ausgeführt wird, nicht effizient. Im Fall der Implementierung (Dblock) kommt hinzu, daß eine Blockgröße von $B = 2N$ verwendet wurde, wodurch u. a. durch die Tile-Schleife Arbeitsmengen von $2B = 4N \approx D = 8\,KB$ und $3B = 6N = 12\,KB$ entstehen, was ein starkes Absinken der Fehlzugriffszahl für Cachegrößen kleiner als 8 KB bzw. 16 KB verhindert. Darüber hinaus entstehen durch die j-Schleife mehrere verhältnismäßig große Arbeitsmengen zwischen $D + 4B - 1 = 32\,KB$ und $D + (s + 4)B - 1 = 59\,KB$.

Vergleich der Implementierungen

Zur Gegenüberstellung der Implementierungen zeigen die Abb. 2.17 für MEDAKZO und 2.18 für BRUSS2D-MIX das Verhalten der Anzahl der Cache-Fehlzugriffe der vier Implementierungen in Abhängigkeit von der Cachegröße, wobei jeweils eine Assoziativität von 16 und verschiedene konstante Zeilengrößen verwendet werden.

Bei Verwendung des Testproblems MEDAKZO wird das Lokalitätsverhalten aufgrund der geringen Zugriffsdistanz überwiegend durch die Schleifenstruktur der Implementierungen bestimmt. Für dieses Testproblem unterscheidet sich die Anzahl der für die einzelnen Implementierungen gemessenen Fehlzugriffe nicht so stark wie bei Verwendung von BRUSS2D-MIX. Insbesondere kann beobachtet werden, daß sich die Anzahl der Fehlzugriffe mit zunehmender Zeilengröße angleicht, da die Implementierungen (A) und (E) von dieser Erhöhung stärker profitieren als (D) und (Dblock). Für sehr kleine Caches mit einer Größe unterhalb von 1 bis 2 KB erzeugt Implementierung (A) weniger Fehlzugriffe als die anderen Implementierungen, da (A) für viele Schleifen nur wenige freie Cachezeilen benötigt, um von der räumlichen Lokalität profitieren zu können. Hinzu kommt, daß die innersten Schleifen von (A), in denen Verfahrenskoeffizienten benutzt werden, nur jeweils einen oder zwei Koeffizienten verwenden, die während des Schleifendurchlaufs konstant sind. Implementierung (E) erzeugt für sehr kleine Cachegrößen eine höhere Zahl von Fehlzugriffen als Implementierung (A). Ist der Cache groß genug bzw. enthält er genug Cachezeilen, um die Verfahrenskoeffizienten zu speichern und jeweils eine Cachezeile für die in den j-Schleifen verwendeten Vektoren bereitzustellen, sinkt die Anzahl der Fehlzugriffe stark ab und unterschreitet aufgrund der besseren Ausnutzung der zeitlichen Lokalität ab einer von der Zeilengröße abhängigen Cachegröße, die in den durchgeführten Experimenten zwischen 1 KB und 32 KB lag, die Anzahl der Fehlzugriffe für Implementierung (A). Da Implementierung (D) die zeitliche Lokalität noch besser ausnutzen kann als Implementierung (E), kann sie eine noch geringere Fehlzugriffszahl erreichen. Dazu war in den durchgeführten Experimenten jedoch eine von der Zeilengröße abhängige Cachegröße von mindestens 4 bis 64 KB erforderlich. Implementierung (Dblock) stellt in den mit MEDAKZO durchgeführten Experimenten einen guten Kompromiß zwischen Implementierung (A) und Implementierung (D) dar. Für kleine Caches erreicht sie nicht ganz die geringe Fehlzugriffszahl von (A), erzeugt jedoch deutlich weniger Fehlzugriffe als (D). Für bestimmte Zeilengrößen gibt es anschließend einen Bereich von Cachegrößen, in dem (Dblock) weniger Fehlzugriffe generiert als sowohl (A) als auch (D). Für größere Caches ist die Fehlzugriffszahl von (Dblock) nur geringfügig höher als die von (D).

Verwendet man das Testproblem BRUSS2D-MIX, verhalten sich die Implementierungen (E) und (D) im Vergleich zueinander sehr ähnlich wie bei Verwendung von MEDAKZO. Durch die höhere Zugriffsdistanz von BRUSS2D-MIX verschlechtert sich jedoch die Anzahl der Fehlzugriffe für (A) und (Dblock) im Vergleich zu (E) und (D) wesentlich. Zwar ist die Anzahl der Fehlzugriffe für beide Implementierungen für

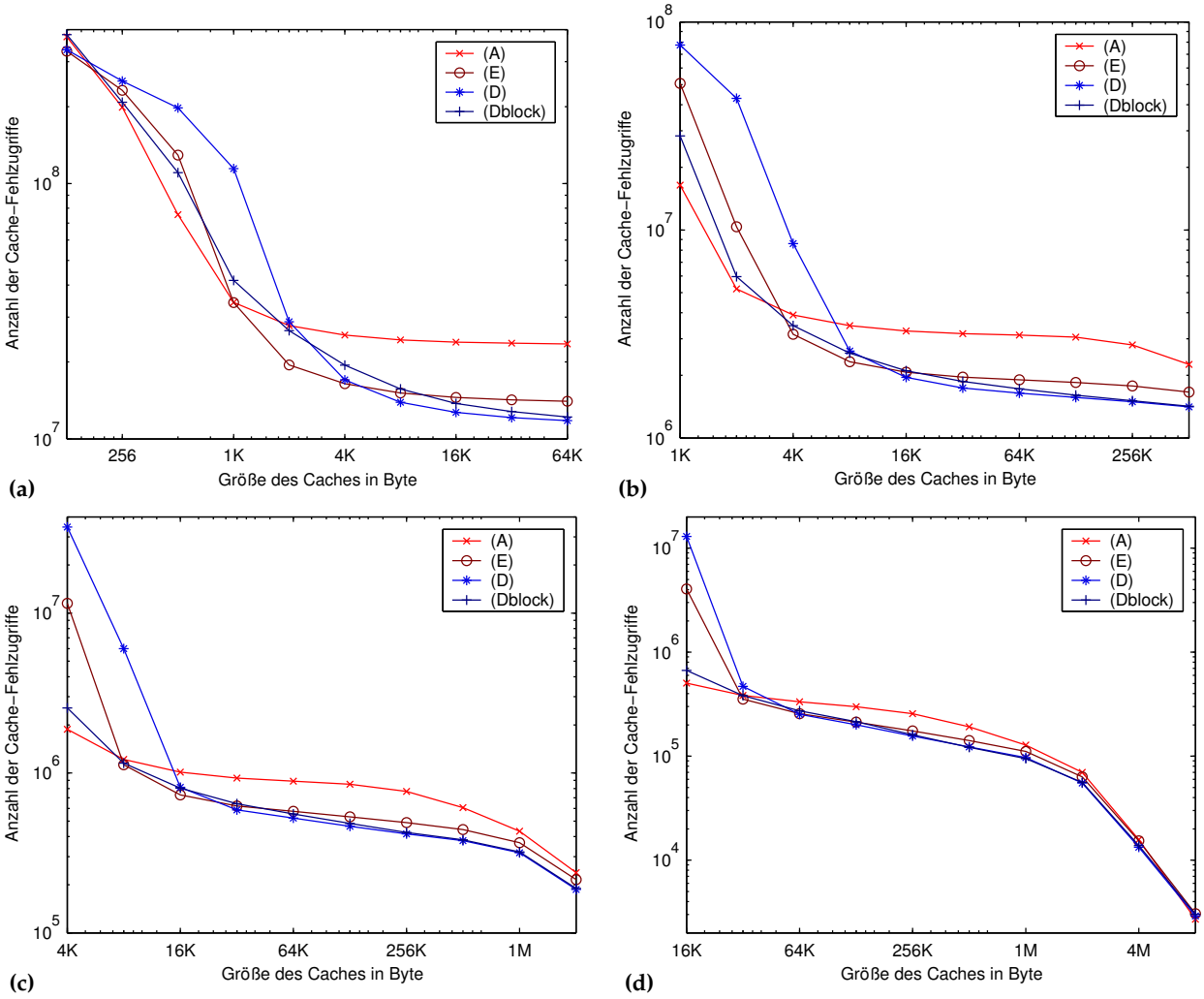


Abb. 2.17: Vergleich der Implementierungen bezüglich der Anzahl der Cache-Fehlzugriffe für MEDAKZO, $N = 15\,000$ bei Verwendung eines 16-fach mengenassoziativen Caches. **(a)** Zeilengröße 8 Byte, **(b)** Zeilengröße 64 Byte, **(c)** Zeilengröße 256 Byte, **(d)** Zeilengröße 1024 Byte.

kleine Cachegrößen noch geringer als für Implementierung (D), Implementierung (E) erzeugt jedoch bereits in diesem Bereich nur eine geringfügig höhere und für höhere Zeilengrößen sogar eine geringere Zahl von Fehlzugriffen als (A) und (Dblock). Implementierung (Dblock) liefert auch für BRUSS2D-MIX einen Kompromiß zwischen (D) und (A): Für kleine Cachegrößen erzeugt sie weniger Fehlzugriffe als (D), jedoch mehr als (A), und für größere Caches verhält sie sich schlechter als (D), aber besser als (A). Allerdings, während (Dblock) für MEDAKZO nur ein geringfügig schlechteres Verhalten aufwies als die jeweils bessere der beiden Implementierungen (A) und (D), nähert sich für BRUSS2D-MIX das Verhalten von (Dblock) mit zunehmender Zeilengröße an das verhältnismäßig schlechte Verhalten von (A) an. Nur für kleine Zeilengrößen kann (Dblock) die geringe Fehlzugriffszahl von (D) erreichen, wenn der Cache ausreichend groß ist. Für Zeilengrößen zwischen 8 Byte und 64 Byte sind hierzu Cachegrößen zwischen 128 KB und 1 MB erforderlich. Diese Cachegrößen reichen aus, um die Arbeitsmengen R_I der Iterationen der j-Schleifen im Cache speichern zu können.

2.4.4 Vergleich des Laufzeitverhaltens

Das Ziel aller vorgestellten Optimierungen des Lokalitätsverhaltens ist die Reduzierung der Laufzeit, damit der Anwender in kürzerer Zeit eine Antwort auf die von ihm untersuchte Fragestellung erhält. Der Erfolg

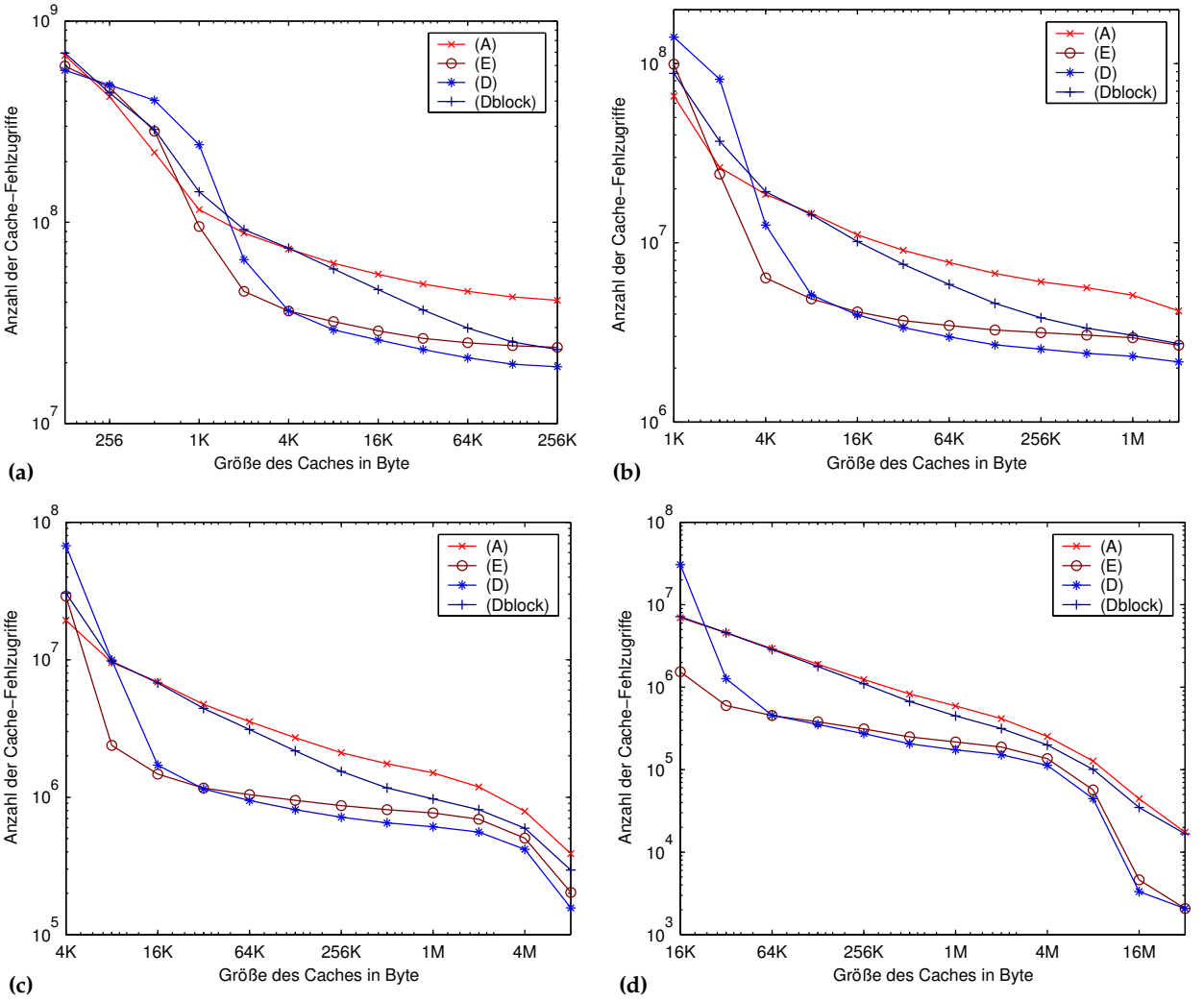


Abb. 2.18: Vergleich der Implementierungen bezüglich der Anzahl der Cache-Fehlzugriffe für BRUSS2D-MIX, $N = 250$ bei Verwendung eines 16-fach mengenassoziativen Caches. **(a)** Zeilengröße 8 Byte, **(b)** Zeilengröße 64 Byte, **(c)** Zeilengröße 256 Byte, **(d)** Zeilengröße 1024 Byte.

der Lokalitätsoptimierungen muß also letzten Endes daran gemessen werden, ob eine geringere Laufzeit erreicht wird. Andere Vergleichskriterien sind dagegen weniger geeignet. Beispielsweise unterscheiden sich die Implementierungen nicht bezüglich der Anzahl der Funktionsauswertungen, und auch die erreichte Genauigkeit ist nahezu identisch, da alle Implementierungen das gleiche numerische Verfahren realisieren und durch Transformationen auf Programmebene auseinander hervorgehen. Auch ein Vergleich der Anzahl der pro Zeiteinheit ausgeführten Gleitkommaoperationen (gemessen in MFLOPS oder GFLOPS) wäre nicht aussagekräftig, da die Laufzeit durch viele weitere Größen bestimmt wird, beispielsweise die Gesamtzahl der Gleitkommaoperationen und auch die Anzahl und die Ausführungsfrequenz der übrigen Instruktionen. Im folgenden Abschnitt wird deshalb eine ausführliche Untersuchung des Laufzeitverhaltens der Implementierungen durchgeführt.

Auswirkung der Arbeitsmengen auf die Laufzeit

Durch die theoretische Analyse in Abschnitt 2.4.2 wissen wir, daß die Implementierungen mehrere unterschiedlich große Arbeitsmengen besitzen. Die anschließend in Abschnitt 2.4.3 beschriebenen Simulationsexperimente haben gezeigt, daß sich diese Arbeitsmengen unterschiedlich stark auf die Anzahl der Fehlzugriffe auswirken und daß es von der Größe des Caches abhängig ist, welche Implementierung die geringste

Hierarchiestufe	Typ	Größe	Assoziativität	Zeilengröße	Latenz
1	Daten	16 KB	4-fach	64 Byte	1 Takt
1	Instruktionen	16 KB	4-fach	64 Byte	1 Takt
2	Unified	256 KB	8-fach	128 Byte	5 Takte
3	Unified	6 MB	12-fach	128 Byte	14 Takte

Tab. 2.4: Cache-Parameter des zur Untersuchung des Laufzeitverhaltens verwendeten Itanium-2-Systems vom Typ HP Integrity rx5670.

Zahl von Fehlzugriffen erzeugt. Es stellt sich nun die Frage, wie sich die Arbeitsmengen auf die Laufzeit auswirken und welche Implementierung unter welchen Bedingungen die geringste Laufzeit besitzt.

Reale Computersysteme besitzen allerdings im Unterschied zum Simulator eine Cache-Hierarchie, bestehend aus ein oder mehreren Caches mit *fester* Größe. Eine Variation der Cachegröße zur Bestimmung der Arbeitsmengen ist deshalb nicht möglich. Man muß daher, z. B. durch Variation der Problemgröße, die Größe der Arbeitsmengen verändern und die daraus resultierende Veränderung des Laufzeitverhaltens untersuchen. Da sich durch die Erhöhung der Problemgröße auch der erforderliche Berechnungsaufwand, d. h. die Anzahl der ausgeführten Zeitschritte und die Anzahl der pro Zeitschritt zu verarbeitenden Komponenten des gewöhnlichen Differentialgleichungssystems vergrößert, ist zur Untersuchung der Arbeitsmengen eine Normierung bezüglich der Zahl der Zeitschritte und der Systemgröße erforderlich. Die aus einer solchen Messung resultierende Funktion beschreibt die Laufzeit einer Implementierung pro Zeitschritt und pro Systemkomponente in Abhängigkeit von der Systemgröße. Wäre keine Speicherhierarchie vorhanden und sind die Kosten einer Funktionsauswertung für eine einzelne Komponente unabhängig von der Systemgröße, würde sich daraus eine nahezu konstante Funktion ergeben, da dann auch die auf diese Weise normierte Laufzeit unabhängig von der Systemgröße wäre. Durch das Vorhandensein von Caches verändert sich jedoch die Laufzeit, wenn sich der Anteil der Cache-Fehlzugriffe erhöht oder verringert. Mit Hilfe der verwendeten Systemgröße und der Stufenzahl sowie gegebenenfalls der benutzten Blockgröße für das Schleifen-Tiling kann man anschließend darauf zurückschließen, welche Arbeitsmenge für die Veränderung der Fehlzugriffszahl verantwortlich ist.

Als Beispiel betrachten wir dazu ein Rechnersystem auf Basis des Itanium-2-Prozessors vom Typ HP Integrity rx5670, das eine interessante dreistufige Cache-Hierarchie aufweist. Die wichtigsten Cache-Parameter dieses Systems sind in Tab. 2.4 zusammengefaßt. Eine genauere Beschreibung dieses Systems wird in Anhang A gegeben. Als Compiler wurde der Intel C/C++ Compiler in der Version 9.0 verwendet.

BRUSS2D. Abbildung 2.19 zeigt einen Vergleich der Laufzeiten pro Zeitschritt und pro Systemkomponente der Implementierungen (A), (E), (D) und (Dblock) für die zwei Testprobleme BRUSS2D-MIX und BRUSS2D-ROW für das Verfahren DOPRI5(4). Für die Implementierung (Dblock) wurde dabei eine Blockgröße von 5120 verwendet, die experimentell bestimmt wurde. Eine detaillierte Diskussion der Bestimmung einer geeigneten Blockgröße wird im nachfolgenden Abschnitt geführt. An dieser Stelle soll zunächst aber der Vergleich der Implementierungen im Vordergrund stehen.

Wie erwartet, ergeben sich durch die Normierung bezüglich der Anzahl der Komponenten und der Anzahl der Zeitschritte Laufzeitkurven, die mit wachsender Systemgröße ansteigen, aber abschnittsweise annähernd konstant verlaufen. Lediglich für kleine Systeme mit wenigen zehn bzw. wenigen hundert Komponenten kann ein Absinken der normierten Laufzeit mit wachsender Systemgröße festgestellt werden, da u. a. die Verbesserung der Ausnutzung der räumlichen Lokalität die Berechnung zusätzlicher Komponenten ohne eine wesentliche Erhöhung der Anzahl der Cache-Fehlzugriffe erlaubt. Wird die Systemgröße jedoch weiter erhöht, steigt auch die normierte Laufzeit etwa in dem Maße, wie sich die Anzahl der Cache-Fehlzugriffe durch das Herausfallen von Arbeitsmengen aus bestimmten Cache-Stufen vergrößert.

Beim Übergang von $N = 90$ auf $N = 91$, d. h. von einer Systemgröße von $n = 16\,200$ auf $n = 16\,562$, weisen die Laufzeitkurven der Implementierungen (D) und (E), aber auch die der Implementierung (A) einen deutlichen Anstieg auf. Eine Untersuchung dieses Effekts hat gezeigt, daß die Ursache dafür ein Strategiewechsel der für die mittels `malloc()` reservierten Speicherblöcke zuständigen Speicherverwaltung ist. Bei Erhöhung von N von 90 auf 91 erhöht sich der Speicherbedarf eines Vektors von ca. 126,6 KB auf 129,4 KB. Ab einer Blockgröße von 128 KB erfolgte die Reservierung der Speicherblöcke durch `malloc()` jedoch in einem anderen Adreßbereich, wobei jeweils die letzten 14 Bit der zurückgelieferten Adressen

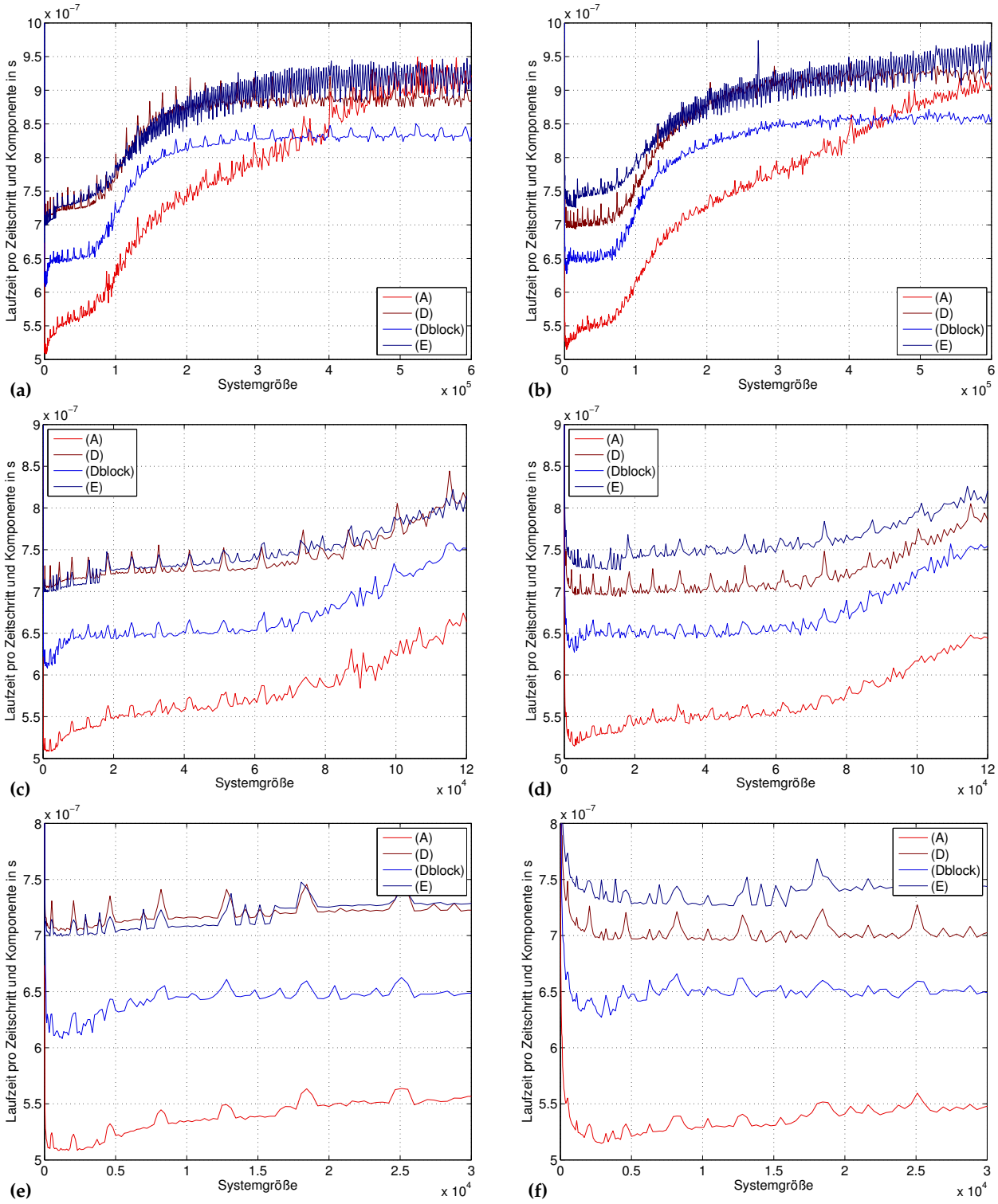


Abb. 2.19: Laufzeit der allgemeinen sequentiellen Implementierungen pro Zeitschritt und Komponente in Abhängigkeit von der Systemgröße auf einem Itanium-2-System. **(a)**, **(c)** und **(e)** BRUSS2D-MIX, **(b)**, **(d)** und **(f)** BRUSS2D-ROW. Für beide Testprobleme wurde das Verfahren DOPRI5(4) verwendet. Die Unterabbildungen zeigen jeweils Ausschnitte der Laufzeitkurve für unterschiedliche Bereiche der Systemgröße.

übereinstimmten. Dies dient vermutlich dem Zweck, eine bessere Zuordnung zu Speicherseiten zu erzielen, deren Größe auf dem betrachteten System auf 16 KB festgelegt war. Eine solche Alignierung wirkt sich jedoch negativ auf die Implementierungen aus, da hierdurch bestimmten Vektorelementen mit gleichem Index zur Speicherung im Cache die gleiche Mengenadresse zugewiesen wird. Dies wiederum führt zu einem Ansteigen der Anzahl der Konfliktfehlzugriffe.

Beim Vergleich der Laufzeiten der unterschiedlichen Implementierungen ist zu beachten, daß diese wesentlich durch Interferenzen zwischen Datenzugriffen und Zugriffen auf den Programmcode beeinflußt werden. Dies wird deutlich, wenn man die Laufzeiten der sequentiellen Implementierungen mit denen der parallelen Implementierungen für gemeinsamen Speicher bei Ausführung mit nur einem Thread vergleicht (siehe Abschnitt 3.4.3). Die Laufzeiten der parallelen Implementierungen unterscheiden sich von den Laufzeiten der sequentiellen Implementierungen teilweise sehr stark. Die Implementierungen selbst unterscheiden sich von den sequentiellen Implementierungen jedoch nur durch wenige zusätzliche Operationen zur Synchronisation mehrerer Threads. Wird nur ein Thread gestartet, ist der resultierende Instruktionsoverhead sehr gering, allerdings verschieben sich durch die zusätzlich eingefügten Operationen die Speicherpositionen der zur Durchführung der Berechnungen erforderlichen Operationen, so daß diese z. T. auf andere Cachezeilen abgebildet werden als bei einer sequentiellen Implementierung. Eine Bedeutung erlangt dies, da der L1-Instruktionscache nur 16 KB umfaßt, der Programmcode der vier untersuchten Implementierungen auf diesem System jedoch eine Größe zwischen 18 KB und 24 KB besitzt. Deshalb muß ein Teil der Zugriffe auf den Programmcode auf niedrigere Ebenen der Speicherhierarchie durchgereicht werden. Dabei kann es zu Interferenzen mit Datenzugriffen kommen. Ursache für die hohe Codegröße ist zum einen die 64-Bit-Architektur des Itanium 2, zum anderen ein hoher Anteil von NOP-Instruktionen (NOP für englisch: *no operation*) am Programmcode. Dieser entsteht dadurch, daß der Itanium-2-Prozessor aufgrund seiner VLIW-Architektur immer jeweils drei Instruktionen gleichzeitig ausführen kann. Aufgrund von Datenabhängigkeiten kann der Compiler diese Instruktionsslots jedoch nicht immer optimal füllen.

Die Analyse der Laufzeitkurven zeigt, daß für den Kurvenverlauf vor allem die zeitliche Wiederverwendung der Stufen- bzw. Argumentvektoren ausschlaggebend ist. Insbesondere die Wiederverwendung vollständiger Vektoren zwischen aufeinanderfolgenden Stufen prägt den Kurvenverlauf stark. Man kann daher für alle Implementierungen beobachten, daß sich der Anstieg der Laufzeitkurve erhöht, sobald der Gesamtspeicherplatzbedarf eines Zeitschrittes aus einer bestimmten Cache-Stufe herauszufallen beginnt. Entsprechend verringert sich der Anstieg, und die normierte Laufzeit stabilisiert sich auf einem höheren Niveau, wenn keine zeitliche Wiederverwendung von Vektoren bzw. bestimmter Vektorteile durch die jeweilige Cache-Stufe mehr möglich ist. Die größte Auswirkung hat dabei der Übergang vom L3-Cache zum Hauptspeicher, da hierbei der größte Anstieg der Speicherzugriffszeiten stattfindet. Besonders stark davon betroffen ist Implementierung (A), deren normierte Laufzeit sich für BRUSS2D-ROW von rund 550 ns für eine Systemgröße von ca. 50 000 auf rund 930 ns für Systemgrößen oberhalb von 800 000 steigert. Dies entspricht einer Verlangsamung um fast 70 %. Der Übergang von L2-Cache auf den L3-Cache führt dagegen nur zu einer geringfügigen Erhöhung der Laufzeit. Am stärksten ist dieser Übergang für Implementierung (A) bei Integration des Testproblems BRUSS2D-MIX ausgeprägt. Hier erhöht sich die normierte Laufzeit von rund 510 ns für eine Systemgröße zwischen 1000 und 3000 auf rund 560 ns für eine Systemgröße von etwa 50 000, was einer Erhöhung um weniger als 10 % entspricht. Ein Übergang vom L1-Cache auf den L2-Cache ist nicht beobachtbar, da der L1-Datencache vom Prozessor nicht für die Zwischenspeicherung von Gleitkommadaten genutzt wird.

Eine genaue Bestimmung der auftretenden Arbeitsmengen ist allerdings nur in wenigen Fällen möglich, da die Laufzeitänderungen nie sprungartig erfolgen, sondern sich über größere Bereiche der Systemgröße erstrecken. Insbesondere, da alle Implementierungen in jeder Stufe eine Schleife mit variabler Grenze ausführen und sich dadurch die Anzahl der in einer Stufe zugegriffenen Vektoren verändert, ergibt sich für jede Stufe eine eigene, charakteristische Arbeitsmenge. Diese führen jedoch nicht zu mehreren kleineren stufenartigen Anstiegen der normierten Laufzeitkurve, sondern es findet ein fließender Übergang statt, da auch Teile der Arbeitsmengen wiederverwendet werden können. Speziell der Übergang vom L2-Cache zum L3-Cache ist schwierig zu untersuchen, da einige Arbeitsmengen erst dann aus dem L2-Cache herausfallen, wenn bereits andere Arbeitsmengen aus dem L3-Cache herausfallen, und sich deshalb verschiedene Effekte überlagern.

Ein Vergleich der Implementierungen zeigt, daß Implementierung (A) aufgrund ihrer guten Ausnutzung der räumlichen Lokalität eine erheblich bessere Laufzeit erreicht als die übrigen Implementierungen,

solange die Systemgröße klein genug ist, so daß mehrere vollständige Vektoren im L3-Cache gespeichert und somit zeitlich wiederverwendet werden können. Ab einer Systemgröße von 393 216 kann der L3-Cache jedoch keine 2 vollständigen Vektoren mehr aufnehmen. Für größere Systeme erreicht daher Implementierung (Dblock) eine bessere Laufzeit, da sie durch ein Tiling der Schleifen über die Systemdimension die zeitliche und räumliche Wiederverwendung von Vektorblöcken ermöglicht.

Im Vergleich zu Implementierung (D) erreicht Implementierung (Dblock) über alle untersuchten Bereiche der Systemgröße hinweg eine geringere Laufzeit, da sie eine bessere Ausnutzung der räumlichen Lokalität ermöglicht. Für beide Implementierungen, (D) und (Dblock), steigt die Laufzeit ab einer Systemgröße von etwa 262 000 nur noch geringfügig an, da keine 3 vollständigen Vektoren mehr im L3-Cache gespeichert werden können, aber mindestens 3 Vektoren benötigt werden, um eine zeitliche Wiederverwendung vollständiger Vektoren zu ermöglichen.

Implementierung (E) erreicht für BRUSS2D-MIX bei kleinen Systemgrößen ungefähr gleiche Laufzeiten wie Implementierung (D), ist allerdings für Systemgrößen über ca. 262 000 meßbar langsamer. Wird BRUSS2D-ROW verwendet, verbessert sich die Laufzeit von (D) gegenüber BRUSS2D-MIX, solange der überwiegende Teil der während eines Zeitschrittes benutzten Daten im L3-Cache gespeichert werden kann. Die Laufzeit von (E) verschlechtert sich dagegen in diesem Bereich. Während die Arbeitsmengen nach und nach aus dem L2-Cache herausfallen, nähern sich die Laufzeiten von (D) und (E) für BRUSS2D-MIX und BRUSS2D-ROW an, stabilisieren sich für die beiden unterschiedlichen Anordnungen der Komponenten jedoch für unterschiedliche Systemgrößen und bei unterschiedlichen Werten. Denn während BRUSS2D-MIX nur eine relativ geringe Zugriffsdistanz von $2N$ aufweist, besitzt BRUSS2D-ROW eine Zugriffsdistanz von N^2. Für BRUSS2D-MIX sind daher für (D) und (E) die erwarteten Arbeitsmengen von $3n$ bzw. $2n$ zu beobachten, wobei (D) nach Herausfallen dieser Arbeitsmengen aus dem L3-Cache eine deutlich bessere Laufzeit als (E) erreicht. Für BRUSS2D-ROW erhöht sich die Laufzeit beider Implementierungen jedoch auch nach dem Herausfallen dieser Arbeitsmengen aus dem L3-Cache weiter, da die durch die Zugriffsdistanz geprägte Arbeitsmenge an Bedeutung gewinnt. Die Laufzeit der Implementierungen (D) stabilisiert sich daher für BRUSS2D-ROW erst, wenn keine 2 Vektoren mehr im L3-Cache gespeichert werden können, und für Implementierung (E) erst, wenn nicht einmal mehr ein Vektor in den L3-Cache paßt. Ab dann ist die Laufzeitdifferenz zwischen (D) und (E) jedoch ähnlich hoch wie für BRUSS2D-MIX.

Das Verhalten von (Dblock) verschlechtert sich durch die Verwendung von BRUSS2D-ROW anstelle von BRUSS2D-MIX für große Systemgrößen in ähnlicher Weise wie für Implementierung (D). Doch im Gegensatz zu Implementierung (D) verschlechtert sich die Laufzeit von (Dblock) für BRUSS2D-ROW auch für kleinere Systemgrößen.

Die Laufzeit der Implementierung (A) erhöht sich durch Verwendung von BRUSS2D-ROW anstelle von BRUSS2D-MIX für sehr kleine Systemgrößen bis ca. 6000 und für sehr große Systemgrößen ab ca. 780 000. In dem dazwischenliegenden Bereich verbessert sich die Laufzeit jedoch deutlich meßbar. Vermutlich führt in diesem Bereich, in dem der überwiegende Teil der in einem Zeitschritt zugegriffenen Daten nicht mehr im L2-Cache, aber noch mindestens ein vollständiger Vektor im L3-Cache gespeichert werden kann, das spezielle Zugriffsmuster von BRUSS2D-ROW zu einer besseren Ausnutzung der räumlichen Lokalität, da ein quasi-paralleler Lauf über zwei um N^2 versetzte Bereiche des Argumentvektors erfolgt.

MEDAKZO. Die für MEDAKZO mit dem Verfahren DOPRI8(7) gemessenen normierten Laufzeiten zeigt Abb. 2.20. Wie für BRUSS2D steigt auch für MEDAKZO die normierte Laufzeit mit wachsender Systemgröße an, da sich die mittleren Kosten der Speicherzugriffsoperationen erhöhen. MEDAKZO besitzt im Unterschied zu BRUSS2D jedoch nur eine sehr kleine Zugriffsdistanz von 2 und eignet sich daher sehr gut, um das Verhalten der Implementierungen nahezu unbeeinflußt durch die Arbeitsmengen des Testproblems zu untersuchen.

Vor allem infolge der höheren Stufenzahl steigt die normierte Laufzeit wesentlich steiler an als in den zuvor betrachteten Beispielen mit BRUSS2D und DOPRI5(4). Ab einer Systemgröße zwischen etwa 40 000 und 50 000 beginnt sich die normierte Laufzeit aller Implementierungen zu erhöhen, da die innerhalb eines Zeitschrittes benutzten Daten nicht mehr vollständig im L3-Cache gespeichert werden können und der L2-Cache keinen einzigen vollständigen Vektor mehr aufnehmen kann. Bereits ab einer Systemgröße von ungefähr 100 000 bis 260 000 stabilisieren sich die Laufzeiten von (D) und (Dblock) und erhöhen sich bis zu einer Systemgröße von etwa 400 000, was einer Arbeitsmenge von $2n$ entspricht, nur noch geringfügig. Danach bleiben sie bis zur höchsten untersuchten Systemgröße von $2 \cdot 10^6$ nahezu konstant bzw. sinken sogar

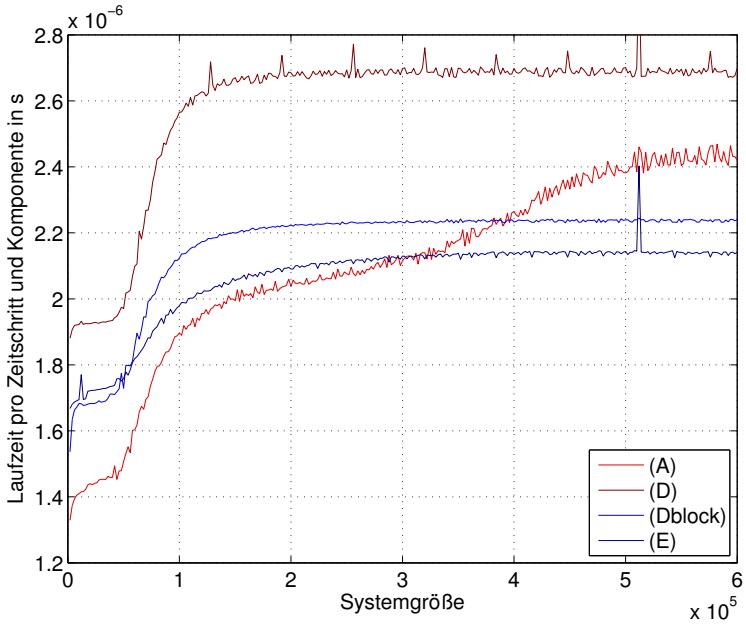


Abb. 2.20: Laufzeit der allgemeinen sequentiellen Implementierungen pro Zeitschritt und Komponente in Abhängigkeit von der Systemgröße auf einem Itanium-2-System für das Testproblem MEDAKZO und das Verfahren DOPRI8(7).

leicht ab. Für die Implementierungen (A) und (E) verringert sich der Anstieg der normierten Laufzeitkurve ebenfalls ab einer Systemgröße von ca. 100 000, allerdings verändert sich der Anstieg langsamer als bei den Implementierungen (D) und (Dblock). Die normierte Laufzeit der Implementierung (E) nähert sich bei einer weiteren Steigerung der Systemgröße bis etwa 450 000 bis 500 000 allmählich einem Maximum und fällt danach bei weiterhin steigender Systemgröße sehr langsam ab. Implementierung (A) besitzt bei einer Systemgröße von etwa 260 000 (entsprechend einer Arbeitsmenge von $3n$) einen Wendepunkt, ab dem sich der Anstieg wieder erhöht, bis die normierte Laufzeit ab einer Systemgröße von ca. 800 000 (entsprechend einer Arbeitsmenge von n) nahezu konstant verläuft.

Wie für BRUSS2D und DOPRI5(4) erreicht Implementierung (A) für kleine Systemgrößen die beste Laufzeit und wird ab einer Systemgröße von ca. 260 000 von Implementierung (Dblock) überholt, welche wiederum für alle Systemgrößen schneller arbeitet als Implementierung (D). Im Unterschied zu den vorherigen Experimenten mit BRUSS2D verschlechtert sich jedoch die relative Performance von Implementierung (D), so daß Implementierung (A) für alle untersuchten Systemgrößen geringere Laufzeiten erzielt. Implementierung (E) profitiert dagegen von der höheren Stufenzahl und übertrifft für Systemgrößen über ca. 50 000 Implementierung (Dblock) und für Systemgrößen über ca. 320 000 auch Implementierung (A).

Auswahl der Blockgröße für das Schleifen-Tiling

Aufgrund der komplexen Abhängigkeiten zwischen dem Zugriffsmuster der Implementierung, dem Zugriffsmuster der Funktionsauswertung und dem Aufbau und der Arbeitsweise der Speicherhierarchie des Zielrechners ist in der Regel eine experimentelle Bestimmung der Blockgröße erforderlich, um einen für die betreffende Kombination aus Implementierung, zu lösendem Problem und Zielplattform optimalen Wert zu finden. Dazu würde man das zu lösende Problem auf der Zielplattform mit unterschiedlichen Blockgrößen integrieren und anschließend die Blockgröße auswählen, die zur geringsten Laufzeit führt. Der Suchraum der möglichen Blockgrößen muß dabei auf geeignete Weise durchlaufen werden.

Abbildung 2.21 zeigt als Beispiel, wie die normierte Laufzeit der Implementierung (Dblock) für die Testprobleme BRUSS2D-MIX und BRUSS2D-ROW bei Verwendung des Verfahrens DOPRI5(4) und der Problemgröße $N = 1000$ auf dem bereits zuvor betrachteten Itanium-2-System von der Blockgröße abhängt. Die Blockgröße wurde dabei mit einer Schrittweite von 1000 erhöht, bis die Systemgröße von $n = 2N^2 = 2 \cdot 10^6$ erreicht wurde. Der Bereich, in dem sich das Laufzeitminimum befindet, wurde anschließend erneut

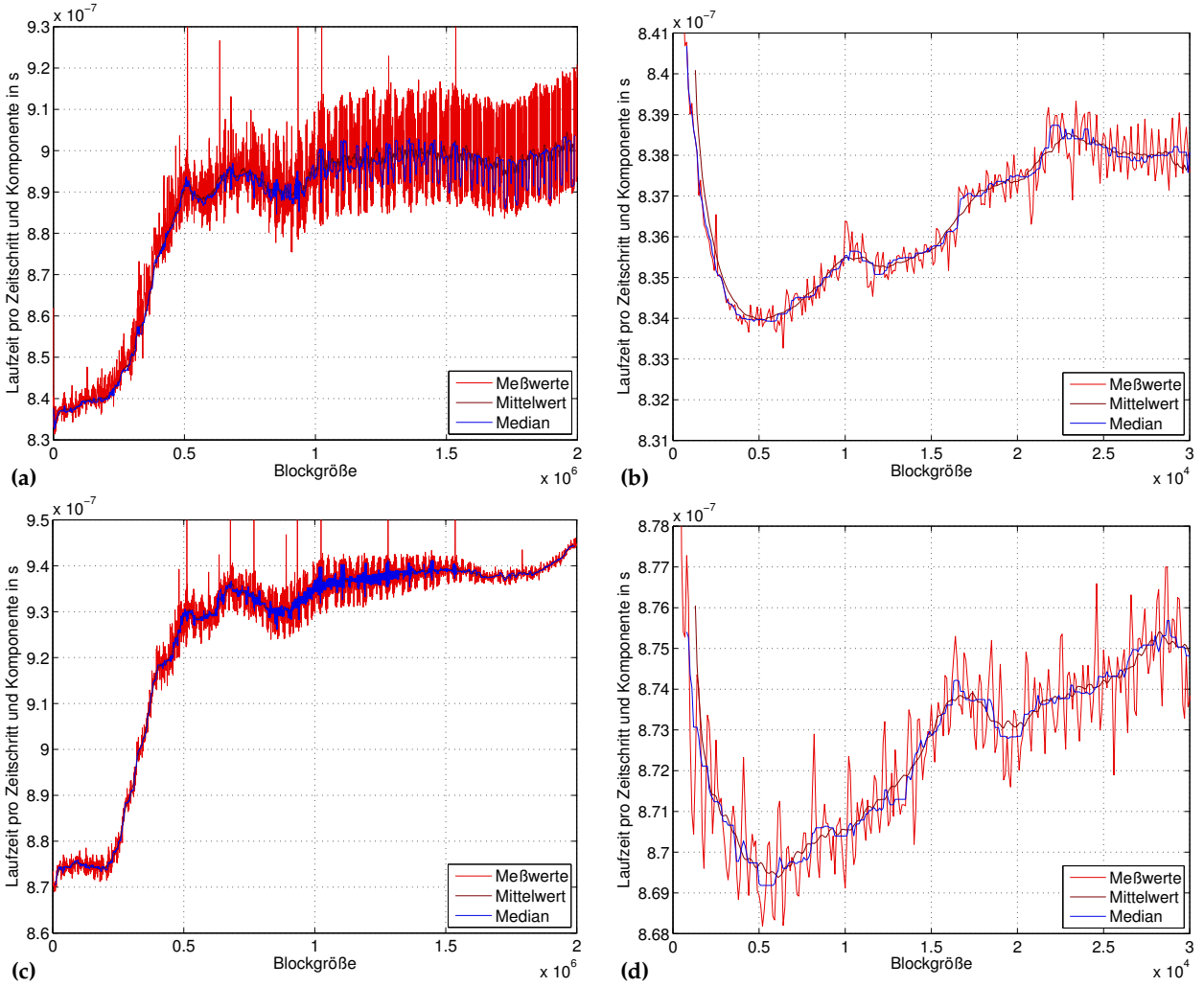


Abb. 2.21: Laufzeit der Implementierung (Dblock) pro Zeitschritt und Komponente in Abhängigkeit von der Blockgröße auf einem Itanium-2-System. **(a)**, **(b)** BRUSS2D-MIX, **(c)**, **(d)** BRUSS2D-ROW. Für beide Testprobleme wurde das Verfahren DOPRI5(4) und die Problemgröße $N = 1000$ verwendet. Die Unterabbildungen zeigen jeweils Ausschnitte der Laufzeitkurve für unterschiedliche Bereiche der Blockgröße.

mit einer feineren Schrittweite von 100 durchlaufen, um die günstigste Blockgröße genauer identifizieren zu können. In die Abbildung wurde zusätzlich zu den gemessenen Laufzeiten ein über je 25 Meßwerte gebildeter Mittelwert sowie ein über je 15 Meßwerte gebildeter Median eingezeichnet, um hochfrequente Laufzeitanteile herauszufiltern und den charakteristischen Kurvenverlauf hervorzuheben.

Die abgebildeten Laufzeitkurven für BRUSS2D-MIX und BRUSS2D-ROW zeigen, daß es bei der Wahl der Blockgröße vor allem darauf ankommt, daß die wichtigsten durch die Blockgröße bestimmten Arbeitsmengen die Größe der niedrigsten Cache-Stufe nicht überschreiten. Für BRUSS2D-ROW läßt sich ein relativ kleiner Bereich der Blockgröße von ca. 241 000 bis ca. 244 000 angeben, der nicht überschritten werden sollte, damit die entscheidende Arbeitsmenge nicht aus dem L3-Cache herausfällt. Für eine Blockgröße aus diesem Bereich kann bereits eine verhältnismäßig gute Laufzeit von rund 878 ns pro Komponente und Zeitschritt erreicht werden. Größere Blockgrößen verschlechtern die normierte Laufzeit relativ stark, so daß diese für Blockgrößen ab 500 000 in der Regel oberhalb von 930 ns liegt, also etwa 5,9 % langsamer ist. Wird die Blockgröße gleich der Systemgröße gesetzt, ergibt sich sogar eine um rund 7,6 % schlechtere normierte Laufzeit von ca. 945 ns. Durch eine günstige Wahl der Blockgröße aus der Umgebung von 5000 läßt sich die normierte Laufzeit noch geringfügig bis unter 870 ns verbessern, da der L2-Cache besser genutzt werden kann. Insgesamt ergibt sich somit im Vergleich zur schlechtesten Blockgröße eine Verbesserung von rund 7,9 %.

Implementierung (D) erreicht jedoch aufgrund der einfacheren Schleifenstruktur eine normierte Laufzeit von nur ca. 921 ns. Der tatsächliche Laufzeitgewinn des Schleifen-Tilings gegenüber Implementierung (D) beträgt also nur rund 5,5%.

Für BRUSS2D-MIX läßt sich die Blockgröße, ab der die entscheidende Arbeitsmenge aus dem L3-Cache herausfällt, weniger scharf eingrenzen, da hier ein anfänglich langsamerer Anstieg der Laufzeit stattfindet. Doch auch hier liegt die kritische Blockgröße in einem ähnlichen Bereich zwischen ungefähr 200 000 und 250 000. Ab einer Blockgröße von ca. 200 000 erhöht sich die normierte Laufzeit mit zunehmendem Anstieg von ca. 840 ns auf ca. 850 ns und steigt dann bis auf über 890 ns ab einer Systemgröße von ca. 500 000. Die günstigste Blockgröße liegt auch für BRUSS2D-MIX bei ungefähr 5000 und ermöglicht eine normierte Laufzeit von rund 834 ns. Im Vergleich zu Implementierung (D), die eine normierte Laufzeit von ca. 889 ns erreicht, entspricht dies einer Verbesserung von rund 6,2 %.

Auch für andere Problemgrößen, deren Gesamtspeicherplatzbedarf oberhalb der Größe des L3-Caches liegt, und auch für das Verfahren DOPRI8(7) sowie das Testproblem MEDAKZO wurden normierte Laufzeitkurven mit ähnlichem Verlauf beobachtet. Insbesondere erwies sich in allen Fällen eine Blockgröße von ungefähr 5000 als günstige Wahl, weshalb für die im vorigen Abschnitt beschriebenen Experimente eine Blockgröße von 5120, entsprechend 40 KB, ausgewählt wurde.

Für die Auswahl einer geeigneten Blockgröße ist es aber in der Regel nicht erforderlich, den gesamten Bereich möglicher Blockgrößen zwischen 1 und der Systemgröße experimentell vollständig zu durchsuchen. Vielmehr lassen sich anhand der Ergebnisse der in Abschnitt 2.4.2 durchgeführten theoretischen Analyse des Lokalitätsverhaltens sowie anhand der Parameter der Cache-Hierarchie des Zielsystems bereits mehrere Intervalle von Blockgrößen bestimmen, die voraussichtlich zu einer guten Laufzeit führen.

Aus Tab. 2.3 kann entnommen werden, daß die von der Blockgröße geprägten Schleifen der Implementierung (Dblock) Arbeitsräume der Größen $2B$, $3B$, $2B+D-1$, $(s+1)B$, $(s-l+1)B$ für $l=2,\ldots,s$, $D+(s-l+4)B-1$ für $l=2,\ldots,s$ sowie $D+(s+4)B-1$. besitzen. Der kleinste dieser Arbeitsräume hat also die Größe B, der größte die Größe $D+(s+4)B-1$. Für eine gegebene Cachegröße C ist daher davon auszugehen, daß die optimale Blockgröße zwischen der Cachegröße C selbst und $(C-D+1)/(s+4)$ liegt. Es genügt also bei einer experimentellen Bestimmung der Blockgröße, diesen Bereich zu durchsuchen. Da Implementierung (Dblock) viele Schleifen mit Arbeitsräumen von $2B$ und $3B$ enthält, wird die ideale Blockgröße jedoch in der Regel kleiner gleich $\frac{1}{2}C$ bzw. $\frac{1}{3}C$ sein, so daß sich der Suchraum weiter eingrenzen läßt. Gilt $D<B$, so genügt die Wahl von $B=\frac{1}{3}C$, um den Arbeitsraum der Größe $2B+D-1\leq 3B$ im Cache halten zu können. Hat D einen größeren Wert, kann eine kleinere Blockgröße vorteilhaft sein. Eine kleinere Blockgröße kann auch eine Wiederverwendung von Cachezeilen zwischen aufeinanderfolgenden Iterationen der äußeren Schleifen über die Systemdimension begünstigen.

Bei mehrstufigen Cache-Hierarchien müssen entsprechend alle Cache-Stufen untersucht werden, um eine optimale Blockgröße zu ermitteln. Auf solchen Systemen kann es sinnvoll sein, ein mehrfaches Schleifen-Tiling zu implementieren, daß jede Cache-Stufe bestmöglich ausnutzt.

Stehen keine Informationen über die vorhandene Cache-Hierarchie zur Verfügung oder soll die Blockgröße architekturunabhängig gewählt werden, besteht die Möglichkeit, sich an der Zugriffsdistanz des zu integrierenden Problems zu orientieren. Zum Beispiel könnte man $B=D=2d(\mathbf{f})+1$ wählen, um Arbeitsmengen mit Größen zwischen $2D$ und $\approx (s+5)D$ zu erzeugen. Dies ist jedoch nur dann sinnvoll, wenn aufgrund der Größe der Zugriffsdistanz des Problems und der erwarteten Cachegröße möglicher Zielsysteme Anlaß zu der Annahme besteht, daß zumindest die meisten dieser Arbeitsmengen im Cache gespeichert werden können und die Blockgröße ausreichend hoch für eine effiziente Wiederverwendung ist.

Die für das Itanium-2-System ermittelte Blockgröße von 40 KB ist etwas kleiner als 1/6 des 256 KB großen L2-Caches. Speziell für BRUSS2D-MIX läßt die normierte Laufzeitkurve durch Wendepunkte bzw. lokale Maxima erkennen, daß auch die Blockgrößen 10 922 $\left(\frac{1}{3}C\right)$ und 16 384 $\left(\frac{1}{2}C\right)$ eine besondere Bedeutung besitzen. Der Bereich zwischen 200 000 und 250 000, ab dem ein Herausfallen aus dem L3-Cache beobachtet wurde, entspricht einer Blockgröße zwischen 1/4 und 1/3 der Kapazität des L3-Caches von 6 MB.

Laufzeit auf verschiedenen Zielsystemen

Zur Ergänzung der auf dem Itanium-2-System durchgeführten detaillierten Untersuchung des Laufzeitverhaltens der Implementierungen wurden mehrere Laufzeitexperimente auf verschiedenen Zielsystemen mit unterschiedlicher Architektur durchgeführt. Tabelle 2.5 zeigt einige für die Testprobleme BRUSS2D-

MIX, BRUSS2D-ROW und MEDAKZO für verschiedene Problemgrößen gemessene Laufzeitwerte. Für die Integration von BRUSS2D wurde dabei das Verfahren DOPRI5(4) und das Integrationsintervall $[0;4]$ verwendet. MEDAKZO wurde mit DOPRI8(7) über dem Intervall $[0;10^{-4}]$ integriert. Da die Länge des Integrationsintervalls das Ergebnis des Vergleichs der Implementierungen nicht beeinflußt, wurden die Integrationsintervalle so gewählt, daß die resultierenden Laufzeiten hinreichend hoch für eine exakte Messung waren, jedoch auch klein genug, um alle Laufzeitmessungen innerhalb einer akzeptablen Zeitspanne abschließen zu können. Für Implementierung (Dblock) wurde darauf verzichtet, auf jedem System eine optimale Blockgröße zu bestimmen. Statt dessen wurde für BRUSS2D-MIX und BRUSS2D-ROW eine Blockgröße von $B = 2N$ und für MEDAKZO eine Blockgröße von $B = 128$ gewählt.

Sun UltraSPARC III Cu. Als erstes vergleichen wir die Laufzeit der Implementierungen auf einer Workstation des Typs Sun Blade 1000, die mit zwei Prozessoren des Typs UltraSPARC III Cu mit einer Taktfrequenz von 1050 MHz ausgestattet ist. Diese besitzt einen 64 KB großen, vierfach mengenassoziativen L1-Datencache, einen 32 KB großen, ebenfalls vierfach mengenassoziativen L1-Instruktionscache sowie einen 8 MB großen, zweifach mengenassoziativen L2-Cache für Daten und Instruktionen. Die Übersetzung der Programme erfolgte mit Hilfe des Sun C Compilers in der Version 5.8. Neben den Laufzeiten wurde zusätzlich mit Hilfe der PCL-Bibliothek (Berrendorf u. Mohr 2003) das Cacheverhalten untersucht. Die ermittelten Zugriffszahlen sind in Tab. 2.6 dargestellt.

Interferenzen der Datenzugriffe mit Zugriffen auf den Programmcode spielen auf dieser Maschine keine so entscheidende Rolle wie auf dem Itanium-2-System, da der Instruktionscache hier die doppelte Kapazität besitzt und gleichzeitig der Binärcode der Implementierungen wesentlich kompakter ist (ca. 10–14 KB).

Für das Testproblem BRUSS2D erreicht für die kleineren Problemgrößen $N = 250$ und $N = 500$ Implementierung (A) die geringste Laufzeit. In beiden Fällen können mindestens zwei vollständige Vektoren im L2-Cache gespeichert werden. Für die Problemgrößen $N = 750$ und $N = 1000$ reicht die Kapazität des L2-Caches jedoch nicht mehr aus, um auch nur einen einzigen Vektor vollständig speichern zu können. In beiden Fällen erreicht (Dblock) eine bessere Laufzeit als (A). Die Implementierungen (D) und (E) sind für alle betrachteten Problemgrößen langsamer als die Implementierungen (A) und (Dblock). (D) und (E) liefern ungefähr gleich hohe Laufzeiten, jedoch ist (E) oft geringfügig schneller. Für BRUSS2D-MIX werden aufgrund der geringeren Zugriffsdistanz bessere Laufzeiten erreicht als für BRUSS2D-ROW. Für MEDAKZO ergibt sich für $N = 30\,000$ die gleiche Reihenfolge der Implementierungen wie für BRUSS2D und kleine Werte für N. Auch hier ist Implementierung (A) am schnellsten, gefolgt von den Implementierungen (Dblock), (E) und (D).

Die Analyse des Zugriffsverhaltens bezüglich des L2-Caches allein genügt nicht zur Erklärung des Laufzeitverhaltens der Implementierungen. Wie Tab. 2.6 zeigt, produzieren die Implementierungen (D) und (Dblock) die geringste Anzahl von Fehlzugriffen auf dem L2-Cache. Die Anzahl der Zugriffe auf den L2-Cache ist jedoch höher als für die Implementierungen (A) und (E). (Dblock) erzeugt im Vergleich zu (D) nahezu die gleiche Anzahl von L2-Cache-Fehlzugriffen, führt aber geringfügig weniger Zugriffe auf diesen Cache aus. Im Vergleich von (A) und (E) erzeugt (A) weniger Zugriffe auf den L2-Cache, generiert aber für BRUSS2D mehr Fehlzugriffe.

Da häufig die Anzahl der L2-Cache-Fehlzugriffe laufzeitbestimmend wirkt, würde man erwarten, daß die Implementierungen (D) und (Dblock) schneller ausgeführt werden können als die anderen Implementierungen und daß beide in etwa die gleiche Laufzeit erreichen. Tatsächlich ist Implementierung (Dblock) deutlich schneller als Implementierung (D) und erreicht für große Problemgrößen sogar eine geringere Laufzeit als Implementierung (A), die in allen betrachteten Fällen schneller als Implementierung (D) ist. Zusätzliche Einzelexperimente haben gezeigt, daß die Ursache dafür darin zu suchen ist, daß die Implementierungen (A) und (Dblock) weniger Instruktionen ausführen und auch weniger Zugriffe auf den L1-Datencache erzeugen, wobei (A) diesbezüglich effizienter arbeitet als (Dblock). Möglich wird dies durch ein Aufrollen der Schleifen über die Systemdimension durch den Compiler, wodurch Vergleiche und Zugriffe auf Schleifenzähler eingespart werden. Darüber hinaus stehen dem Instruktionsscheduler durch das Aufrollen mehr Möglichkeiten zur Verteilung der Instruktionen auf die Funktionseinheiten des Prozessors zur Verfügung.

Intel Pentium 4. Das zweite System, auf dem das Laufzeitverhalten der sequentiellen Implementierungen untersucht wurde, ist eine Workstation des Typs Fujitsu-Siemens SCENIC W600 auf Basis eines Intel-

Testproblem	N	(A)	(D)	(Dblock)	(E)
Sun Blade 1000, Sun UltraSPARC III Cu, 1050 MHz					
BRUSS2D-MIX	250	19,2	28,2	21,5	26,5
BRUSS2D-ROW	250	22,8	31,7	25,0	28,8
BRUSS2D-MIX	500	361,4	602,1	379,3	595,6
BRUSS2D-ROW	500	402,3	653,6	439,0	629,4
BRUSS2D-MIX	750	2108,0	3059,7	1918,0	3092,9
BRUSS2D-ROW	750	2388,0	3443,4	2301,5	3420,5
BRUSS2D-MIX	1000	6771,6	10018,8	6160,3	9906,4
BRUSS2D-ROW	1000	7968,1	11133,8	7310,7	11014,0
MEDAKZO	15000	79,3	116,8	95,3	97,6
MEDAKZO	30000	613,1	981,0	736,6	860,1
Fujitsu-Siemens SCENIC W600, Intel Pentium 4, 3 GHz					
BRUSS2D-MIX	250	9,5	8,7	8,6	7,3
BRUSS2D-ROW	250	9,9	9,6	9,4	8,1
BRUSS2D-MIX	500	144,8	133,8	131,5	109,4
BRUSS2D-ROW	500	147,6	142,4	140,2	121,5
BRUSS2D-MIX	750	704,2	657,9	650,8	549,0
BRUSS2D-ROW	750	738,6	700,8	698,3	596,3
BRUSS2D-MIX	1000	2227,0	2198,4	2061,5	1756,6
BRUSS2D-ROW	1000	2332,4	2288,8	2213,3	1958,2
MEDAKZO	7500	7,0	8,7	8,4	7,1
MEDAKZO	15000	54,5	61,5	56,8	48,5
MEDAKZO	30000	462,0	467,1	428,4	371,2
IBM eServer pSeries 690, IBM Power 4+, 1,7 GHz					
BRUSS2D-MIX	250	7,5	7,3	7,3	6,9
BRUSS2D-ROW	250	8,5	8,7	8,4	7,9
BRUSS2D-MIX	500	125,2	117,1	114,4	106,0
BRUSS2D-ROW	500	136,1	136,4	127,7	117,8
BRUSS2D-MIX	750	587,7	631,4	662,0	577,1
BRUSS2D-ROW	750	725,4	747,2	743,9	628,6
BRUSS2D-MIX	1000	2113,9	1929,2	2265,0	1846,3
BRUSS2D-ROW	1000	2201,8	2242,5	2679,0	2089,9
MEDAKZO	15000	49,9	56,2	57,9	48,5
MEDAKZO	30000	408,2	441,0	438,7	378,8
MEGWARE Saxonid C2, AMD Opteron, 2 GHz					
BRUSS2D-MIX	250	9,5	8,7	8,7	7,7
BRUSS2D-ROW	250	9,8	7,9	9,1	7,9
BRUSS2D-MIX	500	146,0	133,4	133,5	116,9
BRUSS2D-ROW	500	148,5	117,2	134,6	118,2
BRUSS2D-MIX	750	720,4	666,3	672,3	595,4
BRUSS2D-ROW	750	732,8	590,3	678,3	598,1
BRUSS2D-MIX	1000	2300,8	2124,2	2125,3	1831,0
BRUSS2D-ROW	1000	2447,0	1979,0	2275,2	2131,7
MEDAKZO	15000	39,9	50,6	48,9	37,3
MEDAKZO	30000	398,9	440,8	426,5	350,6

Tab. 2.5: Ausführungszeit der allgemeinen sequentiellen Implementierungen auf verschiedenen Rechnersystemen.

Testproblem	N	(A)	(D)	(Dblock)	(E)
Lese-/Schreibzugriffe auf den L2-Cache (L1-Cache-Fehlzugriffe)					
BRUSS2D-MIX	250	$4{,}4 \cdot 10^9$	$5{,}6 \cdot 10^9$	$5{,}5 \cdot 10^9$	$5{,}0 \cdot 10^9$
BRUSS2D-ROW	250	$4{,}2 \cdot 10^9$	$5{,}4 \cdot 10^9$	$5{,}3 \cdot 10^9$	$4{,}8 \cdot 10^9$
BRUSS2D-MIX	500	$6{,}7 \cdot 10^{10}$	$8{,}7 \cdot 10^{10}$	$8{,}4 \cdot 10^{10}$	$7{,}5 \cdot 10^{10}$
BRUSS2D-ROW	500	$6{,}2 \cdot 10^{10}$	$8{,}3 \cdot 10^{10}$	$7{,}9 \cdot 10^{10}$	$7{,}0 \cdot 10^{10}$
BRUSS2D-MIX	750	$3{,}4 \cdot 10^{11}$	$4{,}3 \cdot 10^{11}$	$4{,}2 \cdot 10^{11}$	$3{,}8 \cdot 10^{11}$
BRUSS2D-ROW	750	$3{,}1 \cdot 10^{11}$	$4{,}1 \cdot 10^{11}$	$4{,}0 \cdot 10^{11}$	$3{,}6 \cdot 10^{11}$
BRUSS2D-MIX	1000	$1{,}1 \cdot 10^{12}$	$1{,}5 \cdot 10^{12}$	$1{,}3 \cdot 10^{12}$	$1{,}2 \cdot 10^{12}$
BRUSS2D-ROW	1000	$1{,}0 \cdot 10^{12}$	$1{,}4 \cdot 10^{12}$	$1{,}3 \cdot 10^{12}$	$1{,}1 \cdot 10^{12}$
MEDAKZO	30000	$1{,}7 \cdot 10^{11}$	$2{,}5 \cdot 10^{11}$	$2{,}2 \cdot 10^{11}$	$1{,}9 \cdot 10^{11}$
L2-Cache-Fehlzugriffe					
BRUSS2D-MIX	250	$4{,}9 \cdot 10^7$	$2{,}5 \cdot 10^7$	$2{,}5 \cdot 10^7$	$4{,}2 \cdot 10^7$
BRUSS2D-ROW	250	$5{,}4 \cdot 10^7$	$3{,}0 \cdot 10^7$	$3{,}0 \cdot 10^7$	$4{,}8 \cdot 10^7$
BRUSS2D-MIX	500	$2{,}2 \cdot 10^9$	$1{,}4 \cdot 10^9$	$1{,}4 \cdot 10^9$	$1{,}7 \cdot 10^9$
BRUSS2D-ROW	500	$2{,}2 \cdot 10^9$	$1{,}5 \cdot 10^9$	$1{,}5 \cdot 10^9$	$1{,}7 \cdot 10^9$
BRUSS2D-MIX	750	$1{,}4 \cdot 10^{10}$	$7{,}3 \cdot 10^9$	$7{,}3 \cdot 10^9$	$9{,}4 \cdot 10^9$
BRUSS2D-ROW	750	$1{,}5 \cdot 10^{10}$	$8{,}4 \cdot 10^9$	$8{,}4 \cdot 10^9$	$1{,}0 \cdot 10^{10}$
BRUSS2D-MIX	1000	$4{,}7 \cdot 10^{10}$	$2{,}3 \cdot 10^{10}$	$2{,}3 \cdot 10^{10}$	$3{,}0 \cdot 10^{10}$
BRUSS2D-ROW	1000	$5{,}0 \cdot 10^{10}$	$2{,}7 \cdot 10^{10}$	$2{,}7 \cdot 10^{10}$	$3{,}3 \cdot 10^{10}$
MEDAKZO	30000	$3{,}6 \cdot 10^8$	$3{,}6 \cdot 10^8$	$3{,}5 \cdot 10^8$	$4{,}7 \cdot 10^8$

Tab. 2.6: Anzahl der Cache-Fehlzugriffe der allgemeinen sequentiellen Implementierungen auf einer Sun Blade 1000.

Pentium-4-Prozessors mit einer Taktfrequenz von 3 GHz. Zur Übersetzung wurde hier der Intel C/C++ Compiler 8.0 verwendet. Der L1-Instruktionscache dieses Systems kann 12 K Mikrooperationen speichern. Zwar fehlen Informationen darüber, in wieviele Mikrooperationen die Implementierungen aufgespalten werden, aufgrund der Codegröße von ca. 6 KB für (A), (D) und (E) bzw. 10 KB für (Dblock) und ausgehend davon, daß ein Großteil der Maschineninstruktionen in maximal zwei Mikrooperation überführt wird, ist jedoch anzunehmen, daß es kaum zu Interferenzen von Code und Daten kommt. Für die Zwischenspeicherung von Daten besitzt dieses System einen 8 KB großen, vierfach mengenassoziativen L1-Datencache und einen 512 KB großen, achtfach mengenassoziativen L2-Cache.

Auf dieser Maschine arbeiten die Implementierungen (D) und (E) für die für das Testproblem BRUSS2D betrachteten Problemgrößen deutlich effizienter als Implementierung (A). Nur für die kleineren Problemgrößen, die für das Testproblem MEDAKZO betrachtet wurden, erreicht (A) eine bessere Laufzeit als (D). Dies ist vermutlich auf die geringere Cachegröße und die hohe Assoziativität zurückzuführen, da nur für die für MEDAKZO betrachteten Systemgrößen mindestens ein vollständiger Vektor im L2-Cache gespeichert werden kann. Doch auch bei Verwendung dieses Testproblems ist Implementierung (E) für $N = 15\,000$ und $N = 30\,000$ schneller als (A) und auch schneller als (D) und (Dblock). Auch für alle für BRUSS2D betrachteten Fälle erreicht (E) die beste Laufzeit. (Dblock) erreicht in allen Experimenten eine signifikant geringere Laufzeit als (D). Mit Ausnahme der Experimente mit MEDAKZO mit den beiden kleineren Problemgrößen $N = 15\,000$ und $N = 30\,000$ ist (Dblock) auch schneller als (A).

Die Anzahl der Cache-Fehlzugriffe (Tab. 2.7) gibt ähnlich wie auf der Sun Blade 1000 auch auf dieser Maschine Hinweise auf die Ursachen für das Laufzeitverhalten der Implementierungen. Implementierung (E) erzeugt bezüglich des L1-Caches weniger oder höchstens etwa genau so viele Cache-Fehlzugriffe wie die beste alternative Implementierung und kann deshalb eine bessere Laufzeit erreichen. Auch bezüglich des L2-Caches erzeugt (E) im Vergleich zu (A) und (D) eine geringere oder eine maximal ebenso hohe Fehlzugriffszahl. Nur für MEDAKZO mit $N = 7500$ kann (A) eine geringere Zahl von L2-Cache-Fehlzu-

Testproblem	N	(A)	(D)	(Dblock)	(E)
L1-Datencache					
BRUSS2D-MIX	250	$5{,}0 \cdot 10^8$	$3{,}8 \cdot 10^8$	$4{,}9 \cdot 10^8$	$2{,}1 \cdot 10^8$
BRUSS2D-ROW	250	$6{,}4 \cdot 10^8$	$4{,}5 \cdot 10^8$	$5{,}0 \cdot 10^8$	$2{,}3 \cdot 10^8$
BRUSS2D-MIX	500	$8{,}1 \cdot 10^9$	$8{,}7 \cdot 10^9$	$8{,}2 \cdot 10^9$	$3{,}2 \cdot 10^9$
BRUSS2D-ROW	500	$7{,}9 \cdot 10^9$	$6{,}5 \cdot 10^9$	$8{,}2 \cdot 10^9$	$3{,}6 \cdot 10^9$
BRUSS2D-MIX	750	$3{,}8 \cdot 10^{10}$	$2{,}9 \cdot 10^{10}$	$4{,}2 \cdot 10^{10}$	$1{,}6 \cdot 10^{10}$
BRUSS2D-ROW	750	$4{,}0 \cdot 10^{10}$	$3{,}3 \cdot 10^{10}$	$4{,}2 \cdot 10^{10}$	$1{,}8 \cdot 10^{10}$
BRUSS2D-MIX	1000	$1{,}2 \cdot 10^{11}$	$1{,}5 \cdot 10^{11}$	$1{,}3 \cdot 10^{11}$	$1{,}2 \cdot 10^{11}$
BRUSS2D-ROW	1000	$1{,}3 \cdot 10^{11}$	$1{,}5 \cdot 10^{11}$	$1{,}3 \cdot 10^{11}$	$1{,}3 \cdot 10^{11}$
MEDAKZO	7500	$3{,}2 \cdot 10^8$	$3{,}1 \cdot 10^8$	$3{,}2 \cdot 10^8$	$1{,}3 \cdot 10^8$
MEDAKZO	15000	$2{,}6 \cdot 10^9$	$2{,}2 \cdot 10^9$	$2{,}2 \cdot 10^9$	$9{,}2 \cdot 10^8$
MEDAKZO	30000	$2{,}2 \cdot 10^{10}$	$1{,}8 \cdot 10^{10}$	$2{,}0 \cdot 10^{10}$	$7{,}6 \cdot 10^9$
L2-Cache					
BRUSS2D-MIX	250	$1{,}1 \cdot 10^7$	$1{,}3 \cdot 10^7$	$5{,}8 \cdot 10^6$	$9{,}1 \cdot 10^6$
BRUSS2D-ROW	250	$1{,}1 \cdot 10^7$	$1{,}6 \cdot 10^7$	$6{,}8 \cdot 10^6$	$9{,}6 \cdot 10^6$
BRUSS2D-MIX	500	$1{,}6 \cdot 10^8$	$2{,}0 \cdot 10^8$	$8{,}0 \cdot 10^7$	$1{,}6 \cdot 10^8$
BRUSS2D-ROW	500	$1{,}6 \cdot 10^8$	$2{,}6 \cdot 10^8$	$8{,}7 \cdot 10^7$	$1{,}6 \cdot 10^8$
BRUSS2D-MIX	750	$7{,}4 \cdot 10^8$	$8{,}3 \cdot 10^8$	$3{,}5 \cdot 10^8$	$6{,}8 \cdot 10^8$
BRUSS2D-ROW	750	$7{,}7 \cdot 10^8$	$1{,}2 \cdot 10^9$	$4{,}0 \cdot 10^8$	$7{,}2 \cdot 10^8$
BRUSS2D-MIX	1000	$2{,}3 \cdot 10^9$	$3{,}0 \cdot 10^9$	$1{,}1 \cdot 10^9$	$2{,}2 \cdot 10^9$
BRUSS2D-ROW	1000	$2{,}4 \cdot 10^9$	$3{,}9 \cdot 10^9$	$1{,}2 \cdot 10^9$	$2{,}4 \cdot 10^9$
MEDAKZO	7500	$3{,}2 \cdot 10^6$	$2{,}5 \cdot 10^7$	$1{,}1 \cdot 10^7$	$1{,}9 \cdot 10^7$
MEDAKZO	15000	$4{,}4 \cdot 10^7$	$1{,}8 \cdot 10^8$	$7{,}3 \cdot 10^7$	$1{,}6 \cdot 10^8$
MEDAKZO	30000	$4{,}5 \cdot 10^8$	$1{,}4 \cdot 10^9$	$5{,}7 \cdot 10^8$	$1{,}2 \cdot 10^9$

Tab. 2.7: Anzahl der Cache-Fehlzugriffe der allgemeinen sequentiellen Implementierungen auf einer Pentium-4-Workstation.

griffen erreichen und liefert daher in diesem Fall die beste Laufzeit. Die Verbesserung der Laufzeit von (Dblock) gegenüber (D) resultiert aus einer erheblichen Reduzierung der Anzahl der L2-Cache-Fehlzugriffe. Die Anzahl der Fehlzugriffe auf den nur 8 KB großen L1-Cache steigt dagegen in vielen Fällen an. Dies ist insbesondere bei Verwendung des Testproblems BRUSS2D der Fall, da hier die Blöcke eine Größe zwischen 3,9 KB und 15,6 KB besitzen. Für MEDAKZO beträgt die Blockgröße dagegen nur 1 KB, wodurch die Anzahl der L1-Cache-Fehlzugriffe für dieses Testproblem nur geringfügig ansteigt. Die geringere Laufzeit von (D) gegenüber (A) für BRUSS2D kann nicht ausschließlich durch die Anzahl der Fehlzugriffe erklärt werden, da (D) in allen Beispielen eine höhere Zahl von L2-Cache-Fehlzugriffen generiert und auch die Anzahl der L1-Cache-Fehlzugriffe für $N = 1000$ über der Fehlzugriffszahl von (A) liegt. Für MEDAKZO ist dagegen (A) schneller als (D), obwohl (A) eine höhere Zahl von L1-Cache-Fehlzugriffen erzeugt. Die Zahl der L2-Cache-Fehlzugriffe ist in diesen Experimenten dagegen für (A) deutlich geringer als für (D). Auch auf diesem System wurden für BRUSS2D-MIX jeweils geringere Laufzeiten erreicht als für BRUSS2D-ROW. Dies kann aufgrund der geringeren Anzahl der L2-Cache-Fehlzugriffe auf ein besseres Lokalitätsverhalten, ermöglicht durch die geringere Zugriffsdistanz, zurückgeführt werden.

IBM Power4+. Weitere Laufzeitexperimente wurden auf dem am NIC Jülich installierten Supercomputer JUMP durchgeführt. Dieses System besteht aus mehreren Knoten des Typs IBM eServer pSeries 690, die mit je 32 IBM-Power4+-Prozessorkernen mit einer Taktfrequenz von 1,7 GHz ausgestattet sind. Je zwei Prozessorkerne sind in einen Chip integriert und teilen sich einen 1,5 MB großen, achtfach mengenassoziativen

L2-Cache. Hierdurch kann es bei der Durchführung von Messungen bereits auf der Ebene des L2-Caches zu Störungen durch Prozesse anderer Nutzer kommen. Jedem Prozessorkern stehen jedoch exklusiv nutzbar ein 64 KB großer, direkt abbildender L1-Instruktionscache sowie ein 32 KB großer, zweifach mengenassoziativer L1-Datencache zur Verfügung. Insgesamt ist jeder Knoten mit 512 MB L3-Cache ausgestattet, wobei jedem Chip jeweils ein Modul mit einer Kapazität 32-MB zugeordnet ist. Zur Übersetzung der Programme wurde auf diesem System der IBM C Compiler 6.0.0.9 verwendet.

Ähnlich wie bereits auf der Pentium-4-Workstation erreicht auch hier Implementierung (E) in allen durchgeführten Experimenten die geringste Laufzeit. Implementierung (D) ist in der Mehrzahl der für BRUSS2D-MIX durchgeführten Experimente schneller als Implementierung (A). Für BRUSS2D-ROW und MEDAKZO erreicht jedoch immer Implementierung (A) die bessere Laufzeit. (Dblock) kann für BRUSS2D nur für die kleineren Problemgrößen eine geringere Laufzeit als (D) und (A) erreichen. Bei Verwendung von MEDAKZO ist (Dblock) nur für $N = 30\,000$ geringfügig schneller als (D), jedoch deutlich langsamer als (A). Für BRUSS2D-MIX wurde in allen Experimenten eine geringere Laufzeit als für BRUSS2D-ROW gemessen.

AMD Opteron. Abschließend betrachten wir die Laufzeit der sequentiellen Implementierungen auf einem Cluster-Knoten des Typs MEGWARE Saxonid C2, welcher mit zwei Prozessoren vom Typ AMD Opteron DP 246 ausgestattet ist. Die Prozessoren arbeiten mit einer Taktfrequenz von 2 GHz. Jeder der beiden Prozessoren besitzt je einen separaten 64 KB großen, zweifach mengenassoziativen L1-Cache für Instruktionen und für Daten sowie einen sechzehnfach mengenassoziativen L2-Cache mit einer Kapazität von 1 MB. Auf diesem System erfolgte die Übersetzung der Programme mit der GNU Compiler Collection (GCC) 3.4.3.

Die besten Laufzeiten werden auf diesem System von den Implementierungen (D) und (E) erzielt, wobei (E) für BRUSS2D-MIX und MEDAKZO die geringste Laufzeit erreicht und (D) für BRUSS2D-ROW am schnellsten arbeitet. (Dblock) erzielt für MEDAKZO meßbar geringere Laufzeiten als (D), kann aber die Geschwindigkeit von (A) und (E) nicht erreichen. Für BRUSS2D ist (Dblock) dagegen langsamer als oder höchstens ebenso schnell wie (D). Implementierung (A) liefert in allen Experimenten mit BRUSS2D die schlechteste Laufzeit, ist für MEDAKZO jedoch schneller als (D) und (Dblock).

2.5 Zusammenfassung

Obwohl eingebettete Runge-Kutta-Verfahren im Kern auf einer sehr einfachen Berechnungsvorschrift beruhen, die eine gewichtete Summation von Funktionswerten der rechten Seite des gewöhnlichen Differentialgleichungssystems und des im letzten Zeitschritt bestimmten Approximationswertes beschreibt, stellt eine praktische Realisierung für den Produktionseinsatz hohe Ansprüche, da viele Aspekte beim Entwurf berücksichtigt werden müssen. Neben der Implementierung des eigentlichen Berechnungskernels gilt es unter anderem zu entscheiden, wie die rechte Seite des Differentialgleichungssystems implementiert werden soll, welche Verfahrenskoeffizienten genutzt werden, wie am Anfang und am Ende des Integrationsintervalls verfahren wird, welche Lösungswerte und welche statistischen Informationen ausgegeben werden sollen und wie die Schnittstelle zwischen Integrator und Anwendungsprogramm realisiert wird.

Da Anwender in der Regel an einer schnellen Lösung eines gegebenen Anfangswertproblems interessiert sind, ist die resultierende Ausführungsgeschwindigkeit ein wichtiger Entwurfsaspekt. Dem Entwickler sind hierzu verschiedene Möglichkeiten gegeben, die darauf beruhen, die Anzahl der sequentiell ausgeführten Instruktionen zu verringern oder Wartezeiten bezüglich Operationen und Operanden zu reduzieren. Diesbezüglich kommen verschiedene, z. T. widersprüchliche Optimierungsansätze in Betracht. Einige dieser Techniken, die die Ausführungsgeschwindigkeit entscheidend verbessern können, werden von modernen Compilern beherrscht. Die Auswahl des Compilers und geeigneter Optionen, ist daher eine wichtige Entscheidung in bezug auf die mögliche Ausführungsgeschwindigkeit. Dem Compiler sind bei der Optimierung jedoch durch das durch den Programmierer vorgegebene Quellprogramm enge Grenzen gesetzt. Beispielsweise ist es für den Compiler nicht möglich, alternative Datenstrukturen zu wählen oder Transformationen auf der Ebene der mathematischen Verfahrensvorschrift durchzuführen. Der Entwickler steht somit in der Verantwortung, das Quellprogramm so zu entwerfen, daß nach Optimierung durch den Compiler eine möglichst hohe Ausführungsgeschwindigkeit resultiert.

Moderne Prozessoren erreichen Taktfrequenzen im Gigahertz-Bereich und können teilweise mehrere Instruktionen pro Takt ausführen oder mit speziellen Instruktionen mehrere Operanden parallel verarbeiten. Die Geschwindigkeit des Hauptspeichers liegt jedoch häufig mindestens eine Größenordnung unter der Verarbeitungsgeschwindigkeit des Prozessors, so daß die Effizienz eines Programms überwiegend dadurch bestimmt wird, ob es gelingt, Wartezeiten aufgrund von Speicherzugriffen zu vermeiden oder zumindest weitestgehend zu reduzieren. Dazu ist es notwendig, vorhandene Caches zur Zwischenspeicherung von Daten möglichst effizient auszunutzen, indem die Datenzugriffe so organisiert werden, daß eine hohe räumliche und zeitliche Lokalität resultiert.

Um dies zu verdeutlichen, wurden in diesem Kapitel vier verschiedene Implementierungsmöglichkeiten zur Realisierung des Berechnungskernels vorgestellt und untersucht, die aufgrund unterschiedlicher Schleifenstrukturen zu einem unterschiedlichen Lokalitätsverhalten führen:

1. eine vektororientierte Implementierung, die das Ziel einer möglichst guten Ausnutzung der räumlichen Lokalität verfolgt,
2. eine Implementierung mit dem Ziel der Optimierung der zeitlichen Lokalität bezüglich der Schreibzugriffe auf Argumentvektorelemente,
3. eine Implementierung mit dem Ziel der Optimierung der zeitlichen Lokalität bezüglich der Lesezugriffe auf Stufenvektorelemente sowie
4. eine Implementierung, die – ausgehend von der zuvor genannten Implementierung – durch ein Tiling der Schleifen über die Systemdimension sowohl eine gute Ausnutzung der zeitlichen als auch der räumlichen Lokalität ermöglicht.

Für diese Implementierungsvarianten wurde eine theoretische Analyse der Arbeitsräume der enthaltenen Schleifen durchgeführt und eine Abschätzung der daraus resultierenden Arbeitsmengen vorgenommen, die das unterschiedliche Lokalitätsverhalten der Implementierungen aufzeigen. Hierbei zeigt sich, daß in allen Implementierungen Arbeitsmengen auftreten, die durch das Zugriffsverhalten der Funktionsauswertung der rechten Seite des Differentialgleichungssystems auf den jeweiligen Argumentvektor geprägt sind. Die wichtigste Rolle spielt jedoch die Wiederverwendung vollständiger Vektoren zwischen den Stufen eines Zeitschrittes oder zwischen aufeinanderfolgenden Zeitschritten.

Zur Identifizierung der tatsächlich auftretenden Arbeitsmengen sowie zur Untersuchung der Auswirkung der Assoziativität und der Zeilengröße des Caches auf das Lokalitätsverhalten der Implementierungen wurden Simulationsexperimente durchgeführt, in denen mit Hilfe des Simulators Simics (Magnusson u. a. 2002) verschiedene Arten von Caches mit unterschiedlicher Kapazität, Zeilengröße und Assoziativität nachgebildet wurden und die Anzahl der jeweils auftretenden Cache-Fehlzugriffe ermittelt wurde. Wie die Simulationsergebnisse zeigen, profitieren alle Implementierungen von einer höheren Assoziativität. Sofern keine ungünstige Anordnung der Datenstrukturen vorliegt, führt eine höhere Assoziativität als 4 jedoch nur noch zu einer geringfügigen Verbesserung. Prinzipiell profitieren darüber hinaus alle Implementierungen von einer Erhöhung der Cachezeilengröße, da sie durch die elementweise Iteration über Vektoren eine hohe räumliche Lokalität aufweisen. Allerdings führt die Erhöhung der Zeilengröße auch zu einer Vergrößerung der auftretenden Arbeitsmengen, was für bestimmte Cachegrößen zu einer Verschlechterung der Fehlzugriffszahl führt. Die beobachteten Arbeitsmengen decken sich im wesentlichen mit den aus der theoretischen Analyse entstandenen Erwartungen. Insbesondere zeigt sich, daß vor allem die zeitliche Wiederverwendung vollständiger Vektoren das Lokalitätsverhalten bestimmt, während andere Arbeitsmengen in der Regel wenig signifikant sind.

Zur Untersuchung der Auswirkung des Lokalitätsverhaltens der Implementierungen auf die Laufzeit wurden Experimenten auf verschiedenen Zielsystemen mit unterschiedlicher Architektur durchgeführt. Auf einem Rechnersystem auf Basis des Itanium-2-Prozessors mit einer dreistufigen Cache-Hierarchie wurde eine detaillierte Analyse der auftretenden Arbeitsmengen durchgeführt, indem die Laufzeit pro Systemkomponente und pro Zeitschritt in Abhängigkeit von der Systemgröße gemessen wurde. Erhöht sich diese normierte Laufzeit ab einer bestimmten Systemgröße, ist dies ein Hinweis auf das Herausfallen einer Arbeitsmenge aus einer Cache-Stufe. Die auf diese Weise beobachteten Laufzeitkurven werden überwiegend durch das Herausfallen von Vektoren aus dem L3-Cache geprägt. Jedoch hat auch das Zugriffsverhalten des betrachteten Testproblems einen entscheidenden Einfluß. Für kleine Problemgrößen, die die Speicherung

von mindestens einem oder zwei vollständigen Vektoren im L3-Cache gestatten, erreicht die vektororientierte Implementierung durch ihre gute Ausnutzung der räumlichen Lokalität die beste Laufzeit. Größere Probleme können mit anderen Implementierungen effizienter integriert werden. Für das Testproblem BRUSS2D liefert in diesem Fall das Schleifen-Tiling die geringste Laufzeit. Für das Testproblem MEDAKZO mit einer sehr geringen Zugriffsdistanz ist dagegen die Ausnutzung der zeitlichen Lokalität bezüglich der Schreibzugriffe auf Argumentvektorelemente erfolgreicher.

Weitere Laufzeitexperimente wurden für ausgewählten Problemgrößen auf vier unterschiedlichen Zielsystemen auf Basis der Prozessoren Sun UltraSPARC III Cu, Intel Pentium 4, IBM Power4+ und AMD Opteron durchgeführt. Wie auf dem Itanium-2-System war auch auf dem UltraSPARC-III-System die vektororientierte Implementierung für kleine Problemgrößen am erfolgreichsten. Für größere Probleme, für die die Speicherung vollständiger Vektoren im L2-Cache nicht mehr möglich war, konnte durch das Schleifen-Tiling eine bessere Laufzeit als für die vektororientierte Implementierung erreicht werden. Auf diesem System wurde auch der größte Laufzeitunterschied zwischen den betrachteten Implementierungsvarianten beobachtet. So konnte für das Testproblem BRUSS2D-MIX bei Verwendung der Problemgröße $N = 1000$ durch Verwendung des Schleifen-Tilings anstelle von dessen Ausgangsimplementierung die Laufzeit um über 38 % von 2 h 47 min auf 1 h 43 min verbessert werden. Auf dem Pentium-4-System sowie auf dem Power4+-System ist die Ausnutzung der zeitlichen Lokalität bezüglich der Schreibzugriffe auf Argumentvektorelemente am erfolgreichsten. Diese Implementierungsvariante ist auch auf dem Opteron-System sehr erfolgreich. Für BRUSS2D-ROW führt jedoch die Ausnutzung der zeitlichen Lokalität bezüglich der Lesezugriffe auf Stufenvektorelemente zu einer geringeren Laufzeit. Bei der Interpretation dieser Ergebnisse ist zu beachten, daß auf diesen Maschinen darauf verzichtet wurde, eine optimale Blockgröße für das Schleifen-Tiling zu bestimmen, so daß nicht sichergestellt ist, daß die für diese Implementierung gemessenen Laufzeiten das bestmögliche Verhalten dieser Implementierung repräsentieren.

Insgesamt bestätigen die Laufzeitexperimente, daß die Optimierung des Lokalitätsverhaltens entscheidend für die effiziente Ausnutzung moderner Prozessoren und das Erreichen einer möglichst geringen Laufzeit ist. Unterschiedliche Eigenschaften der Zielplattformen und des Zugriffsverhaltens der zu integrierenden Probleme sowie unterschiedliche Programmtransformationen bei Verwendung verschiedener Compiler führen jedoch dazu, daß keine Implementierungsvariante unter allen Umständen die beste Laufzeit liefert. Vielmehr ist für jede Zielplattform und für jedes zu integrierende Problem eine erneute Evaluierung und Optimierung erforderlich, um die jeweilige Zielplattform bestmöglich auszunutzen.

3 Parallele Implementierung eingebetteter Runge-Kutta-Verfahren

Das vorausgehende Kapitel hat deutlich gemacht, welches Potential die Optimierung der Speicherzugriffslokalität zur Laufzeitverbesserung sequentieller Implementierungen eingebetteter Runge-Kutta-Verfahren bietet. Gegenstand dieses Kapitels sind Möglichkeiten zur parallelen Ausführung eingebetteter Runge-Kutta-Verfahren und die Untersuchung verschiedener Ansätze zur Verbesserung der parallelen Effizienz. Der wichtigste Ansatzpunkt dabei ist die Optimierung des Lokalitätsverhaltens, da dies insbesondere auf parallelen Rechnersystemen mit gemeinsamem Adreßraum zu einer signifikanten Verbesserung der Laufzeit führen kann. Aber auch Ansätze zur dynamischen Lastbalancierung werden diskutiert, die helfen können, Wartezeiten aufgrund von Synchronisationsoperationen zu reduzieren.

3.1 Parallelitätspotential eingebetteter Runge-Kutta-Verfahren

Bereits in Abschnitt 1.2 wurden verschiedene Ansätze zur parallelen Lösung von Anfangswertproblemen vorgestellt. In diesem Zusammenhang wurden in Abschnitt 1.2.1 auch Ansätze zur Parallelisierung von Runge-Kutta-Verfahren besprochen. Wie Arbeiten verschiedener Autoren, u. a. die Arbeiten von Miranker u. Liniger (1967), Khalaf u. Hutchinson (1992), Iserles u. Nørsett (1990) und Jackson u. Nørsett (1995) zeigen, besitzen explizite Runge-Kutta-Verfahren nur ein sehr eingeschränktes Parallelitätspotential bezüglich der Methode. Aufgrund der durch die Verfahrensvorschrift (2.2) gegebenen Abhängigkeiten ist im Falle einer vollbesetzten Verfahrensmatrix A eine sequentielle Berechnung der Stufen erforderlich. Um eine parallele Berechnung der Stufen zu ermöglichen, müssen deshalb zur Entkoppelung der Stufen spezielle Verfahrensmatrizen konstruiert werden, die an mehreren Positionen mit 0 besetzt sind. Durch diese Einschränkung sind derartige Verfahren allerdings nicht in der Lage, ebenso gute numerische Eigenschaften zu erreichen wie Verfahren mit vollbesetzten Verfahrensmatrizen. Insbesondere ist durch Satz 1.1 eine Schranke für die maximal erreichbare Ordnung gegeben. Setzt man jedoch eine vollbesetzte Verfahrensmatrix voraus, so beschränkt sich das verfügbare Parallelitätspotential bezüglich der Methode auf die Instruktionsparallelität, d. h. die parallele Ausführung weniger Maschineninstruktionen durch verschiedene Funktionseinheiten eines Prozessors.

Eine Parallelisierung bezüglich der Zeitschritte kommt nicht in Betracht, da die erste Stufe jedes Zeitschrittes von der letzten Stufe des vorausgehenden Zeitschrittes abhängig ist. Insbesondere bei Verwendung einer adaptiven Schrittweitenkontrolle kann die erste Stufe eines Zeitschrittes erst berechnet werden, nachdem entschieden wurde, ob der vorausgehende Zeitschritt akzeptiert oder verworfen werden soll, und nachdem eine neue Schrittweite festgelegt wurde. Eine Parallelisierung bezüglich der Zeitschritte ist daher nur mit Waveform-Relaxationsverfahren möglich, indem das parallel berechnete Integrationsintervall mehrfach durchlaufen wird.

Das größte Parallelitätspotential eingebetteter Runge-Kutta-Verfahren besteht in der Parallelität bezüglich des Differentialgleichungssystems. Denn zumindest theoretisch erlaubt es die Verfahrensvorschrift (2.2), daß alle n Komponenten der Stufenvektoren $\mathbf{v}_1, \ldots, \mathbf{v}_s$ bzw. der Argumentvektoren $\mathbf{w}_2, \ldots, \mathbf{w}_s$ unabhängig voneinander berechnet werden. Der verfügbare Parallelitätsgrad hängt somit direkt mit der Größe des Differentialgleichungssystems zusammen, d. h., um n_P Prozessoren nutzen zu können, muß das gewöhnliche Differentialgleichungssystem mindestens die Dimension n_P besitzen. Zwar können aus diesem Grund Systeme mit kleiner Dimension weniger stark beschleunigt werden als größere Systeme, eine parallele Ausführung ist jedoch vor allem für rechenintensive Probleme interessant, die entweder eine große

Dimension besitzen oder deren Funktionsauswertungen f_j für die einzelnen Komponenten des Differentialgleichungssystems aufwendig zu berechnen sind. Für Systeme kleiner Dimension mit teuren Funktionsauswertungen f_j kommt zusätzlich zur Ausnutzung der Parallelität bezüglich der Systemkomponenten eine Parallelisierung der einzelnen Komponentenfunktionen in Frage. Ist eine Parallelisierung der Komponentenfunktionen jedoch nicht möglich, weil z. B. die Anzahl der ausgeführten Instruktionen zu gering ist, bleibt für Systeme kleiner Dimension nur der Ausweg der parallelen Ausführung der Zeitschritte mit Hilfe von Waveform-Relaxations-Techniken. Es existieren jedoch viele gewöhnliche Differentialgleichungssysteme, die einen hohen Parallelitätsgrad bezüglich der Komponenten besitzen. Insbesondere durch eine Semi-Diskretisierung partieller Differentialgleichungssysteme können sehr große gewöhnliche Differentialgleichungssysteme entstehen, deren Dimension um so größer ist, je feiner die räumliche Diskretisierung vorgenommen wird.

Es liegt daher nahe, für eine parallele Implementierung eingebetteter Runge-Kutta-Verfahren in erster Linie das Parallelitätspotential bezüglich der Komponenten des Differentialgleichungssystems auszunutzen. Jedoch zeigt sich in der Praxis, daß es aufgrund der vorhandenen Abhängigkeiten, d. h. der Koppelung zwischen den Stufen sowie der für die Schrittweitenkontrolle notwendigen Akkumulation des Näherungswertes für den lokalen Fehler, schwierig ist, dieses Potential vollständig auszuschöpfen. So konnten etwa in Laufzeitexperimenten, die von Rauber u. Rünger (1999b) auf zwei Parallelrechnern mit verteiltem Speicher, einer Intel Paragon und einer IBM SP2, durchgeführt wurden, gute Speedup-Werte aufgrund der hohen Kommunikationskosten nur unter der Voraussetzung erzielt werden, daß die Funktionsauswertungszeiten der Komponentenfunktionen f_j sehr hoch waren.

Um das vorhandene Parallelitätspotential besser ausnutzen zu können, sind deshalb weiterführende Untersuchungen des Skalierbarkeitsverhaltens erforderlich, die als Ausgangspunkt für Optimierungen und die Erstellung effizienterer Implementierungen dienen können. In diesem Zusammenhang werden in den nachfolgenden Abschnitten dieses Kapitels Implementierungsmöglichkeiten für verteilten und gemeinsamen Speicher sowie für hybride Speicherarchitekturen vorgestellt und diskutiert. Dabei wird insbesondere das Skalierbarkeitsverhalten der Implementierungen in Abhängigkeit von der Speicherzugriffslokalität betrachtet. Als weiterer Ansatz zur Optimierung wird in Abschnitt 3.6 die Verwendung von Lastbalancierungstechniken in Betracht gezogen. In Kapitel 4 werden anschließend Möglichkeiten zur Reduzierung der Abhängigkeiten zwischen den Stufen durch Ausnutzung spezifischer Eigenschaften des Differentialgleichungssystems untersucht.

3.2 Entwicklung eines parallelen Rahmenprogramms

Eine Parallelisierung bezüglich der Komponenten des gewöhnlichen Differentialgleichungssystems kann für jede der in Abschnitt 2.4 vorgestellten Implementierungsvarianten und auch für viele alternative Realisierungen des Berechnungskernel durchgeführt werden, indem die Iterationen der Schleifen über die Systemdimension parallel durch mehrere Kontrollflüsse ausgeführt werden. Dazu ist es zunächst erforderlich, eine geeignete Verteilung der Komponenten bzw. der zu den Komponenten korrespondierenden Schleifeniterationen festzulegen. Zusätzlich müssen geeignete Synchronisationsoperationen eingefügt werden, die sicherstellen, daß die aus der parallelen Ausführung hervorgehende Berechnungsreihenfolge in Übereinstimmung mit den durch die Berechnungsvorschrift (2.2) gegebenen Datenabhängigkeiten steht.

3.2.1 Wahl der Datenverteilung

Da die Komponentenfunktionen f_j unabhängig voneinander ausgeführt werden können und deshalb keine der Schleifen über die Systemdimension Abhängigkeiten zwischen den Iterationen der Schleifen aufweist, ist prinzipiell eine beliebige Verteilung der Komponenten auf die beteiligten Kontrollflüsse möglich. Die Wahl der Datenverteilung kann sich jedoch auf das Lokalitätsverhalten und den zur Synchronisation der Kontrollflüsse erforderlichen Aufwand auswirken. In Frage kommen z. B. typische Datenverteilungen, wie sie in vielen Anwendungen aus dem wissenschaftlichen Rechnen für die Verteilung eindimensionale Felder verwendet werden (vgl. Rauber u. Rünger 2000). Alle diese Unterteilungen haben zum Ziel, jedem der n_K Kontrollflüsse etwa den gleichen Arbeitsaufwand zuzuordnen, indem die Indexmenge der Komponenten $I = \{j \in \mathbb{N} : 1 \leq j \leq n\}$ in n_K annähernd gleich große Teilmengen I_k, $k = 1, \ldots, n_K$, partitioniert wird.

- **Blockweise Verteilung:** Die Menge I wird durch Festlegung von $n_K - 1$ Intervallgrenzen

$$j_1 < j_2 < \cdots < j_{n_K - 1} \tag{3.1}$$

in n_K disjunkte Intervalle (Blöcke)

$$I_k = (j_{k-1}, j_k] = \{j \in I : j_{k-1} < j \leq j_k\} \quad \text{für} \quad k = 1, \ldots, n_K \tag{3.2}$$

unterteilt, wobei $j_0 = 0$ und $j_{n_K} = n$ gesetzt wird. Zur Festlegung der Intervallgrenzen kommen verschiedene Vorgehensweisen in Frage. Der einfachste Fall liegt vor, wenn n ein ganzzahliges Vielfaches von n_K ist. Dann können jedem Kontrollfluß genau n/n_K Komponenten zugeordnet werden, indem $j_k = k \cdot n/n_K$ gesetzt wird. Ist n kein ganzzahliges Vielfaches von n_K, legt man die Intervallgrenzen so fest, daß $(n \bmod n_K)$ Kontrollflüsse $\lceil n/n_K \rceil$ Komponenten erhalten und allen verbleibenden Kontrollflüssen $\lfloor n/n_K \rfloor$ Komponenten zugeordnet werden, z. B.:

$$j_k = \begin{cases} k \cdot \left\lceil \frac{n}{n_K} \right\rceil & \text{für } k \leq (n \bmod n_K), \\ (n \bmod n_K) \cdot \left\lceil \frac{n}{n_K} \right\rceil + (k - (n \bmod n_K)) \cdot \left\lfloor \frac{n}{n_K} \right\rfloor & \text{sonst.} \end{cases} \tag{3.3}$$

- **Zyklische Verteilung:** Die Elemente der Indexmenge I werden reihum an die beteiligten Kontrollflüsse verteilt, z. B.:

$$I_k = \{j \in I : (j-1) \bmod n_K = k - 1\} \quad \text{für} \quad k = 1, \ldots, n_K \ . \tag{3.4}$$

- **Blockzyklische Verteilung:** Ähnlich einer blockweisen Verteilung wird die Indexmenge I zunächst in $n_B \gg n_K$ Intervalle (Blöcke)

$$B_b = (j_{b-1}, j_b] = \{j \in I : j_{b-1} < j \leq j_b\} \quad \text{für} \quad b = 1, \ldots, n_B \tag{3.5}$$

mit den Intervallgrenzen

$$0 = j_0 < j_1 < j_2 < \cdots < j_{n_B - 1} < j_{n_B} = n \tag{3.6}$$

zerlegt. Diese Blöcke werden anschließend zyklisch den beteiligten Kontrollflüssen zugeordnet:

$$I_k = \bigcup_{\substack{b=1,\ldots,n_B \\ (b-1) \bmod n_K = k-1}} B_b \quad \text{für} \quad k = 1, \ldots, n_K \ . \tag{3.7}$$

Um eine möglichst hohe Lokalität der Speicherzugriffe bei der parallelen Implementierung eines eingebetteten Runge-Kutta-Verfahrens zu erreichen, ist in der Regel eine blockweise Datenverteilung die beste Wahl. Da jedem Kontrollfluß ein einzelner zusammenhängender Block von Komponenten zugeordnet wird, ist in diesem Fall die bestmögliche Ausnutzung von zeitlicher und räumlicher Lokalität zwischen den Iterationen der Schleifen über die Systemdimension gegeben, wenn der Block komponentenweise durchlaufen wird. Auch für eine effiziente Realisierung des Datenaustauschs ist eine blockweise Verteilung besonders günstig. Eine ausführlichere Diskussion der Vorteile einer blockweisen Verteilung wird später im Zusammenhang mit der Erstellung von Implementierung für spezifische Speicherarchitekturen durchgeführt.

3.2.2 Synchronisation der Schrittweitenkontrolle

Während des Integrationsvorgangs, d. h. während des Durchlaufens des Integrationsintervalls müssen die Zeitschritte von den beteiligten Kontrollflüssen synchron ausgeführt werden. Das bedeutet, zu jedem Zeitpunkt berechnen alle Kontrollflüsse jeweils den gleichen Zeitschritt und verwenden dazu eine einheitliche Schrittweite. Insbesondere muß eine einheitliche Entscheidung darüber getroffen werden, ob ein Zeitschritt akzeptiert oder verworfen werden soll.

Zur Durchführung der Schrittweitenkontrolle muß zunächst die Approximation des lokalen Fehlers

$$\epsilon = \frac{1}{\mathrm{TOL}} \max_{j=1,\ldots,n} \left| \frac{\mathbf{e}[j]}{\mathbf{s}[j]} \right| \tag{3.8}$$

berechnet werden. Da jedem Kontrollfluß k aufgrund der Verteilung der Vektorkomponenten nur die durch die Indexmenge I_k bestimmten Komponenten der Vektoren $\mathbf{e}$ und $\mathbf{s}$ bekannt sind, bietet es sich an, daß alle Kontrollflüsse parallel jeweils ihr lokales Maximum $\epsilon^k_{\text{local}}$ bestimmen:

$$\epsilon^k_{\text{local}} = \frac{1}{\text{TOL}} \max_{j \in I_k} \left| \frac{\mathbf{e}[j]}{\mathbf{s}[j]} \right| \quad . \tag{3.9}$$

Anschließend kann das globale Maximum aller Kontrollflüsse

$$\epsilon = \max_{k=1,\ldots,n_K} \epsilon^k_{\text{local}} \tag{3.10}$$

bestimmt werden, wozu allerdings ein Datenaustausch zwischen den Kontrollflüssen erforderlich ist. Zwei mögliche Vorgehensweisen kommen in Betracht:

1. Nur ein einzelner Kontrollfluß kennt das globale Maximum ϵ. Dieser Kontrollfluß führt die Schrittweitenkontrolle allein durch, d. h., er entscheidet allein, ob der Schritt akzeptiert oder verworfen wird, und bestimmt die neue Schrittweite. Das Ergebnis der Entscheidung sowie die neue Schrittweite wird anschließend den anderen Kontrollflüssen bekannt gegeben, die in der Zwischenzeit auf das Eintreffen dieser Daten gewartet haben.

2. Alle Kontrollflüsse kennen das globale Maximum ϵ. Anstatt zu warten, können die Kontrollflüsse jetzt die Schrittweitenkontrolle gleichzeitig durchführen. Da alle Kontrollflüsse vom gleichen Wert für ϵ ausgehen, treffen sie die gleiche Entscheidung bezüglich des Akzeptierens oder Verwerfens des Zeitschrittes und berechnen einen identischen Wert für die neue Schrittweite.

Zur Realisierung der ersten Vorgehensweise müssen an zwei Programmstellen Daten ausgetauscht werden. Eine *Einzelakkumulationsoperation* (auch: *Reduktionsoperation*) wird benötigt, um anhand der von den einzelnen Kontrollflüssen berechneten lokalen Maxima $\epsilon^k_{\text{local}}$ das globale Maximum ϵ zu bestimmen und dem ausgewählten Kontrollfluß zu übermitteln. Später wird die neue Schrittweite sowie die Information, ob der Zeitschritt akzeptiert oder verworfen wird, mit Hilfe einer Broadcastoperation an die übrigen Kontrollflüsse weitergegeben. Die zweite Vorgehensweise erfordert lediglich an einer Programmstelle einen Datenaustausch. Hier wird mittels einer spezialisierten Variante einer *Multiakkumulationsoperation* das globale Maximum ϵ bestimmt und allen Kontrollflüssen bekannt gegeben.

Welche Vorgehensweise effizienter ist, hängt im wesentlichen von der Realisierung des Datenaustauschs ab. Welche Möglichkeiten es dazu gibt, ist wiederum abhängig von der Speicherarchitektur und dem Aufbau des Verbindungsnetzwerks. Wird die benötigte Funktionalität der Multiakkumulationsoperation beispielsweise durch eine Einzelakkumulationsoperation und eine Broadcastoperation implementiert, werden sich die resultierenden Laufzeiten beider Vorgehensweisen in der Regel nur unwesentlich unterscheiden. Alle in dieser Arbeit betrachteten Implementierungen nutzen die zweite Vorgehensweise.

3.2.3 Realisierung des Rahmenprogramms

Aus den zuvor beschriebenen Überlegungen ergibt sich das in Abb. 3.1 gezeigte Rahmenprogramm. Das Rahmenprogramm verwendet das sogenannte SPMD-Programmiermodell (englisch: *single program multiple data*), d. h., alle beteiligten Kontrollflüsse führen das gleiche Programm aus. Anhand der Eingabedaten, beispielsweise der ID des Kontrollflusses, können jedoch gegebenenfalls unterschiedliche Verzweigungen ausgeführt werden, was für die Realisierung des Rahmenprogramms jedoch nicht erforderlich ist.

Es wird eine blockweise Datenverteilung verwendet, die in Zeile 3 des Rahmenprogramms berechnet wird. Dazu bestimmt jeder Kontrollfluß k den Index der ersten Komponente $j_{\text{first}} = j_{k-1} + 1$ und den Index der letzten Komponente $j_{\text{last}} = j_k$ seines Datenbereichs $I_k = (j_{k-1}, j_k]$ nach der Vorschrift (3.3). Nach Festlegung der Datenverteilung wird die Integration vorbereitet, d. h., benötigter Speicher wird allokiert und die Anfangsschrittweite wird bestimmt. Danach wird das Integrationsintervall durch Ausführung der Zeitschritte durchlaufen. Abschließend gibt der Integrator den nicht mehr benötigten Speicher frei und kehrt zum aufrufenden Programm zurück. Welche Datenstrukturen verwendet und wie sie realisiert werden, hängt von der konkreten Implementierung und von der Speicherarchitektur der Zielplattform ab.

```
1:  void parallel_embedded_rk_integrator(double t_0, double t_e, double η[])
2:  {
3:      Bestimme erste Komponente, j_first, und letzte Komponente, j_last, des lokalen Datenbereichs;

4:      Allokiere Speicher für benötigte Vektoren;
5:      Berechne Startwert für Schrittweite h;

6:      for (t := t_0; t < t_e; h := min {h, t_e − t})      // Laufe über das Integrationsintervall, bis t_e erreicht ist
7:      {
8:          Berechne lokale Komponenten von v_1, ..., v_s bzw. w_2, ..., w_s;
9:          Berechne lokale Komponenten von Δη, e und s;

10:         ε_local := 1/TOL max_{j:=j_first,...,j_last} |e[j]/s[j]|;    // Berechne Maximum der lokalen Fehlerkomponenten

11:         ε := multiaccumulation(max, ε_local);                // Berechne Approximation des lokalen Fehlers

12:         if (ε ≤ 1.0)                                                  // Zeitschritt wird akzeptiert
13:         {
14:             η[j] += Δη[j] für j := j_first, ..., j_last;          // Aktualisiere lokale Komponenten von η
15:             t += h;                                                   // Aktualisiere t

16:             h := neue Schrittweite für nächsten Zeitschritt;
17:         }
18:         else                                                          // Zeitschritt wird verworfen
19:             h := neue Schrittweite für Wiederholung des Zeitschrittes;
20:     }

21:     Gib nicht mehr benötigten Speicher frei;
22: }
```

Abb. 3.1: Rahmenprogramm eines Zeitschrittes eines parallelen eingebetteten Runge-Kutta-Verfahrens, an dem sich alle in dieser Arbeit vorgestellten parallelen Implementierungen orientieren.

Die Zeitschritte bestehen wie bei den sequentiellen Implementierungen aus dem Berechnungskernel (Zeile 8 und 9) und der Schrittweitenkontrolle (Zeile 10 bis 19), die wie in Abschnitt 3.2.2 beschrieben durchgeführt wird. Bei der Vorstellung konkreter paralleler Implementierungen in den nachfolgenden Abschnitten wird nur der Berechnungskernel der jeweiligen Implementierung dargestellt, da die übrigen Programmabschnitte der Implementierungen im wesentlichen durch das Rahmenprogramm vorgegeben sind und somit die Realisierung des Berechnungskernels ausschlaggebend für die Laufzeitunterschiede ist.

3.3 Parallelrechner mit verteiltem Adreßraum

Parallele Rechnersysteme mit einem verteilten Adreßraum besitzen im Vergleich zu Systemen mit gemeinsamem Adreßraum eine einfachere Hardwarearchitektur, da es nicht erforderlich ist, Cache-Kohärenz und Speicherkonsistenz global für alle Recheneinheiten zu gewährleisten. Infolgedessen besetzen Parallelrechner mit verteiltem Adreßraum momentan sowohl das Niedrigpreissegment der Supercomputer als auch die Spitze der von Meuer u. a. gepflegten Top-500-Liste. Denn einerseits kann man sehr preiswerte Parallelrechner mit verteiltem Adreßraum, sogenannte Clustersysteme, aus Standardkomponenten aufbauen, wie sie teilweise auch in handelsüblichen Workstations zum Einsatz kommen (vgl. Anhang A). Auf der anderen Seite ist es möglich, auf der Basis eines verteilten Adreßraumes technisch ausgefeilte parallele Rechnersysteme zu konstruieren, in denen eine sehr große Anzahl von Prozessoren effizient zusammenarbeiten kann.

So ist der derzeit[1] leistungsfähigste Supercomputer ein Parallelrechner mit verteiltem Adreßraum von Typ Blue Gene/L (Gara u. a. 2005), der mit 131 072 Prozessoren ausgestattet ist.

Zur Synchronisation der auf den Prozessoren eines Parallelrechners mit verteiltem Adreßraum ausgeführten Kontrollflüsse werden über das Verbindungsnetzwerk Nachrichten ausgetauscht. Der Nachrichtenaustausch (englisch: *message passing*) ist daher das native Programmiermodell für diese Art von Parallelrechnern. Die meisten modernen Parallelrechner bieten dem Programmierer zur Realisierung des Nachrichtenaustausches eine Anwendungsschnittstelle, die sich am MPI-Standard (für englisch: *message passing interface*; MPI Forum 1994, 1995, 1997) orientiert. Die im MPI-Standard definierten Funktionen werden zu diesem Zweck vom Hersteller in Form einer Programmbibliothek zur Verfügung gestellt. Neben den herstellerspezifischen MPI-Bibliotheken existieren auch mehrere Open-Source-Projekte zur Entwicklung von MPI-Implementierungen, z. B. MPICH (Gropp u. a. 1996), dessen Nachfolgeprojekt MPICH2 (Gropp 2002) sowie LAM/MPI (Burns u. a. 1994; Squyres u. Lumsdaine 2003) und dessen Nachfolgeprojekt Open MPI (Gabriel u. a. 2004; Graham u. a. 2006), die jeweils eine Vielzahl von Plattformen unterstützen. Parallele Programme, die MPI zur Kommunikation nutzen, werden in der Regel im SPMD-Stil programmiert. Die parallele Ausführung erfolgt, indem auf jedem Prozessor für das resultierende Binärprogramm ein Kontrollfluß in Form eines Betriebssystemprozesses erzeugt wird.

3.3.1 Implementierung für verteilten Adreßraum

Die Implementierung eingebetteter Runge-Kutta-Verfahren für Parallelrechner mit verteiltem Adreßraum erfordert an mehreren Programmstellen einen Datenaustausch. Zur Realisierung der Schrittweitenkontrolle gibt das im vorhergehenden Abschnitt entwickelte Rahmenprogramm die Verwendung einer Multiakkumulationsoperation zur Berechnung der Approximation des lokalen Fehlers vor. Jeder beteiligte Kontrollfluß versendet dabei das von ihm zuvor berechnete lokale Maximum der ihm zugeordneten Fehlerkomponenten und empfängt das ermittelte globale Maximum der von allen Kontrollflüssen versandten Nachrichten. Der MPI-Standard stellt für einen solchen Datenaustausch die Funktion `MPI_Allreduce()` zur Verfügung. Diese Funktion arbeitet blockierend, d. h., sie kehrt erst dann zum aufrufenden Programm zurück, wenn das akkumulierte Ergebnis empfangen wurde.

Innerhalb des Berechnungskernels ist ebenfalls ein Austausch von Daten notwendig, da aufgrund der Datenverteilung jeder Kontrollfluß zunächst nur die Teilmenge der Komponenten der Argumentvektoren $\mathbf{w}_1, \ldots, \mathbf{w}_s$ kennt, die er selbst berechnet hat. In jeder Stufe muß deshalb ein Datenaustausch durchgeführt werden, um für die Auswertung der Funktion $\mathbf{f}$, die die rechte Seite des Differentialgleichungssystems bestimmt, den jeweils benötigten Argumentvektor allen beteiligten Kontrollflüssen bekannt zu machen. Für diesen Datenaustausch versendet jeder Kontrollfluß den von ihm berechneten Teil der Komponenten des auszutauschenden Argumentvektors in Form einer Nachricht und empfängt eine Nachricht, die alle Komponenten des Arguments enthält. Eine Kommunikationsoperation, die von mehreren Kontrollflüssen abgesendete Nachrichten zu einer Gesamtnachricht verkettet und bei der jeder beteiligte Kontrollfluß die Gesamtnachricht empfängt, wird als *Multibroadcastoperation* bezeichnet. Der MPI-Standard enthält zur Realisierung einer Multibroadcastoperation die Funktion `MPI_Allgather()` für den Fall, daß alle von den Kontrollflüssen versandten Nachrichten die gleiche Größe besitzen, sowie die Funktion `MPI_Allgatherv()` für den Fall unterschiedlich großer Nachrichten. Beide MPI-Operationen arbeiten blockierend.

Die genannten MPI-Operationen erwarten die zu versendende Nachricht in einem zusammenhängenden Speicherbereich, dessen Größe und Anfangsadresse der MPI-Operation beim Aufruf übergeben wird. Für die zu empfangende Nachricht muß die Anwendung einen ausreichend großen Speicherbereich vorhalten, dessen Anfangsadresse ebenfalls an die MPI-Operation übergeben wird. Aus diesem Grund ist die Verwendung einer blockweisen Datenverteilung vorteilhaft, da in diesem Fall die zu versendenden Komponenten der Argumentvektoren bereits in einem zusammenhängenden Speicherbereich abgelegt sind. Andernfalls wäre es entweder erforderlich, die ausgehende Nachricht durch Kopieren der zu versendenden Daten in einen zu diesem Zweck angelegten Sendepuffer zusammenzustellen und in umgekehrter Weise mit der empfangenen Nachricht zu verfahren oder den Datenaustausch in kleinere Nachrichten aufzusplitten und mehrere Aufrufe von Kommunikationsoperationen durchzuführen. Dies würde jedoch zu einer

[1] Top-500-Liste vom November 2006

Vergrößerung des Instruktionsoverheads und in der Regel auch zu einer weniger effizienten Kommunikation führen.

Wir betrachten im folgenden parallele Versionen der zwei Implementierungsvarianten (A) und (D), deren Berechnungskernel in den Abb. 3.2 und 3.3 dargestellt sind. Sie entsprechen weitestgehend den zugehörigen sequentiellen Implementierungen. Jedoch wurde zur blockweisen Verteilung der Komponenten des gewöhnlichen Differentialgleichungssystems der Indexbereich der Schleifen über die Systemdimension auf das Intervall $[j_{\text{first}}, j_{\text{last}}]$ eingeschränkt. Weiterhin wird in jeder Stufe vor der Ausführung der Funktionsauswertungen der für die Funktionsauswertungen benötigte Argumentvektor (η bzw. $\mathbf{w}$) mit Hilfe einer Multibroadcastoperation allen beteiligten Kontrollflüssen bekannt gemacht.

3.3.2 Theoretische Analyse der Skalierbarkeit

Die Tatsache, daß bei der Programmierung für verteilten Adreßraum mit MPI oder ähnlichen Anwendungsschnittstellen alle Kommunikationsoperationen explizit im Programmcode formuliert werden, ermöglicht eine relativ einfache Abschätzung der Kommunikationskosten (vgl. Rauber u. Rünger 1999b). Die Implementierungen (A) und (D) führen in jeder Stufe eine Allgather-Operation sowie zur Schrittweitenkontrolle eine Allreduce-Operation aus. Der Zeitaufwand für die Kommunikation, der von der Anzahl der beteiligten Prozessoren und der Größe des Differentialgleichungssystems abhängig ist, läßt sich folglich abschätzen als:

$$T_p^{\text{comm}}(n) = sT_{\text{allgather}}(p, n/p) + T_{\text{allreduce}}(p, 1) \ . \tag{3.11}$$

$T_{\text{allgather}}(p, n/p)$ ist dabei die Zeit, die zur Ausführung einer Allgather-Operation benötigt wird, an der p Prozessoren beteiligt sind und wobei jeder Prozessor eine Nachricht, bestehend aus n/p Fließkommawerten, versendet. $T_{\text{allreduce}}(p, 1)$ ist die Zeit, die p Prozessoren für die Ausführung einer Allreduce-Operation benötigen, wobei die Nachrichten nur aus einem einzelnen Fließkommawert bestehen.

Die Ausführungszeiten der Kommunikationsoperationen, die sich für eine bestimmte Anzahl von Prozessoren p und eine bestimmte Nachrichtengröße m ergeben, sind abhängig von der Architektur des verwendeten Parallelrechners. Für spezielle Topologien des zugrundeliegenden Verbindungsnetzwerks lassen sich jedoch asymptotische Schranken für die Anzahl der Übertragungsschritte angeben (vgl. Rauber u. Rünger 2000). Die Anzahl der Übertragungsschritte für die Kommunikationsoperationen „Allgather“ und „Allreduce“ auf verschiedenen Topologien statischer Netzwerke zeigt Tab. 3.1. Die Anzahl der Übertragungsschritte ist dabei in Abhängigkeit von der durch das Netzwerk verbundenen Anzahl von Prozessoren angegeben, da diese die Anzahl der Übertragungsschritte bestimmt. In die Laufzeit geht zusätzlich die Nachrichtengröße m linear ein, da in jedem Übertragungsschritt die Nachricht, bestehend aus m Datenelementen, über eine Kante des Netzwerks übertragen wird. Ausgehend von dieser Annahme setzen wir

$$T_{\text{allgather}}(p, m) = mT_{\text{allgather}}(p) \tag{3.12}$$

und

$$T_{\text{allreduce}}(p, m) = mT_{\text{allreduce}}(p) \ , \tag{3.13}$$

wobei $T_{\text{allgather}}(p)$ und $T_{\text{allreduce}}(p)$ die Anzahl der benötigten Übertragungsschritte bezeichnet. Die Kommunikationszeit beträgt somit:

$$T_p^{\text{comm}}(n) = s\frac{n}{p}T_{\text{allgather}}(p) + T_{\text{allreduce}}(p) \ . \tag{3.14}$$

Zur Einschätzung der Skalierbarkeit betrachten wir den *Speedup*, $S_p(n)$, und die *parallele Effizienz*, $E_p(n)$, der Implementierungen. Der Speedup berechnet sich aus dem Verhältnis der Laufzeit der besten sequentiellen Implementierung, $T_{\text{seq}}(n)$, und der Laufzeit der parallelen Implementierung bei Verwendung von p Prozessoren, $T_p(n)$:

$$S_p(n) = \frac{T_{\text{seq}}(n)}{T_p(n)} \ . \tag{3.15}$$

Die parallele Effizienz bezeichnet das Verhältnis der Laufzeit der besten sequentiellen Implementierung zu den *parallelen Kosten*

$$C_p(n) = p \cdot T_p(n) \ . \tag{3.16}$$

```
1:  // Berechne Stufenvektor v_1
2:  multibroadcast(η);
3:  for (j := j_first; j ≤ j_last; j++) v_1[j] := f_j(t + c_1 h, η);
4:  // Berechne Stufenvektoren v_2, ..., v_s
5:  for (l := 2; l ≤ s; l++)
6:  {
7:      for (j := j_first; j ≤ j_last; j++) w[j] := a_l1 v_1[j];
8:      for (i := 2; i < l; i++)
9:          for (j := j_first; j ≤ j_last; j++) w[j] += a_li v_i[j];
10:     for (j := j_first; j ≤ j_last; j++) w[j] := η[j] + h w[j];
11:     multibroadcast(w);
12:     for (j := j_first; j ≤ j_last; j++) v_l[j] := f_j(t + c_l h, w);
13: }
14: // Berechne Δη und Fehlervektor e
15: for (j := j_first; j ≤ j_last; j++)
16: {
17:     Δη[j] := b_1 v_1[j];
18:     e[j] := b̃_1 v_1[j];
19: }
20: for (l := 2; l ≤ s; l++)
21:     for (j := j_first; j ≤ j_last; j++)
22:     {
23:         Δη[j] += b_l v_l[j];
24:         e[j] += b̃_l v_l[j];
25:     }
26: // Berechne Skalierungsfaktoren
27: for (j := j_first; j ≤ j_last; j++) s[j] := |η[j]| + |h · v_1[j]|;
```

Abb. 3.2: Parallele Version der Implementierung (A) für verteilten Adreßraum.

```
1:  // Berechne Stufenvektor v_1
2:  multibroadcast(η);
3:  for (j := j_first; j ≤ j_last; j++)
4:  {
5:      ṽ := h f_j(t + c_1 h, η);
6:      s[j] := |η[j]| + |ṽ|;
7:      for (i := 2; i ≤ s; i++) w_i[j] := η[j] + a_i1 ṽ;
8:      Δη[j] := b_1 ṽ;
9:      e[j] := b̃_1 ṽ;
10: }
11: // Berechne Stufenvektoren v_2, ..., v_s
12: for (l := 2; l ≤ s; l++)
13: {
14:     multibroadcast(w_l);
15:     for (j := j_first; j ≤ j_last; j++)
16:     {
17:         ṽ := h f_j(t + c_l h, w_l);
18:         for (i := l + 1; i ≤ s; i++) w_i[j] += a_il ṽ;
19:         Δη[j] += b_l ṽ;
20:         e[j] += b̃_l ṽ;
21:     }
22: }
```

Abb. 3.3: Parallele Version der Implementierung (D) für verteilten Adreßraum.

Für die parallele Effizienz ergibt sich somit:

$$E_p(n) = \frac{T_{\text{seq}}(n)}{C_p(n)} = \frac{T_{\text{seq}}(n)}{p \cdot T_p(n)} = \frac{S_p(n)}{p} \quad . \tag{3.17}$$

Um den eingabeabhängigen Berechnungsaufwand zu beschreiben, wird die Größe des Differentialgleichungssystems, n, als Parameter verwendet. Ein ideales paralleles Programm hätte eine parallele Laufzeit von $T_p(n) = T_{\text{seq}}(n)/p$ und folglich einen Speedup von p und eine parallele Effizienz von 1. Die meisten realistischen Programme erreichen diese idealen Werte jedoch nur selten, weil sie z. B. einen gewissen sequentiellen Codeanteil enthalten oder ein bestimmter Zeitaufwand zur Durchführung von Kommunikation zwischen den Prozessoren erforderlich ist.

Basierend auf der Annahme, daß der Berechnungsaufwand wesentlich durch die Funktionsauswertungen der Komponentenfunktionen f_j bestimmt wird und alle Funktionsauswertungen etwa die gleiche Berechnungszeit benötigen, benutzt man als Grundlage zur Abschätzung der Laufzeit der sequentiellen und parallelen Implementierung die zur Auswertung einer Komponentenfunktion erforderliche Zeit $T_f(n)$, die von der Systemgröße abhängig sein kann. Da in jeder der s Stufen für jede der n Komponenten eine Funktionsauswertung durchgeführt werden muß, ergibt sich die sequentielle Laufzeit als

$$T_{\text{seq}}(n) = snT_f(n) \quad . \tag{3.18}$$

Netzwerktopologie	Allgather	Allreduce
vollständiger Graph	$\Theta(1)$	$\Theta(1)$
lineares Feld	$\Theta(p)$	$\Theta(p)$
Baum	$\Theta(p)$	$\Theta(\log p)$
d-dimensionales Gitter	$\Theta(p)$	$\Theta(\sqrt[d]{p})$
d-dimensionaler Hyperwürfel	$\Theta(p/\log p)$	$\Theta(\log p)$

Tab. 3.1: Asymptotische Schranken für die Anzahl der Übertragungsschritte der für die Implementierung eingebetteter Runge-Kutta-Verfahren benötigten Kommunikationsoperationen auf verschiedenen Topologien statischer Netzwerke in Abhängigkeit von der Anzahl der Prozessoren, p, die die Knoten des jeweiligen Netzwerks bilden.

Da im Fall der parallelen Implementierung jeder Prozessor in jeder Stufe nur n/p Komponenten auswertet, beträgt der parallele Berechnungsaufwand

$$T_p^{\text{comp}}(n) = s\frac{n}{p}T_f(n) = \frac{1}{p}T_{\text{seq}}(n) \ . \tag{3.19}$$

Die parallele Laufzeit ergibt sich dementsprechend als Summe des Berechnungs- und des Kommunikationsaufwandes als

$$\begin{aligned} T_p(n) &= T_p^{\text{comp}}(n) + T_p^{\text{comm}}(n) \\ &= s\frac{n}{p}T_f(n) + s\frac{n}{p}T_{\text{allgather}}(p) + T_{\text{allreduce}}(p) \ , \end{aligned} \tag{3.20}$$

und wir erhalten durch Einsetzen den Speedup

$$\begin{aligned} S_p(n) = \frac{T_{\text{seq}}(n)}{T_p(n)} &= \frac{snT_f(n)}{s\frac{n}{p}T_f(n) + s\frac{n}{p}T_{\text{allgather}}(p) + T_{\text{allreduce}}(p)} \\ &= \frac{1}{\frac{1}{p} + \frac{1}{pT_f(n)}T_{\text{allgather}}(p) + \frac{1}{snT_f(n)}T_{\text{allreduce}}(p)} \end{aligned} \tag{3.21}$$

und die parallele Effizienz

$$E_p(n) = \frac{S_p(n)}{p} = \frac{1}{1 + \frac{1}{T_f(n)}T_{\text{allgather}}(p) + \frac{p}{snT_f(n)}T_{\text{allreduce}}(p)} \ . \tag{3.22}$$

Aus diesen Funktionen, die das Verhalten des Speedups und der Effizienz in Abhängigkeit von p, n und s beschreiben, lassen sich bereits einige Aussagen über die Skalierbarkeit direkt ablesen. So ist ersichtlich, daß der ideale Speedup von p bzw. die ideale Effizienz von 1 aufgrund des erforderlichen Kommunikationsaufwandes nicht erreicht werden kann. Speedup und Effizienz verbessern sich jedoch, wenn s, n oder $T_f(n)$ erhöht werden. Insbesondere die Vergrößerung von $T_f(n)$ verbessert das Verhältnis zwischen Berechnungsaufwand und Kommunikationskosten und erhöht somit gleichzeitig Speedup und Effizienz. Die Vergrößerung von s und n wirkt sich ebenfalls positiv auf dieses Verhältnis aus, verringert jedoch lediglich den negativen Einfluß der Allreduce-Operation; der Einfluß der Allgather-Operation bleibt unverändert. Allerdings wird die Laufzeit der Allgather-Operation mit dem Faktor $1/p$ multipliziert, so daß sich bei einer Erhöhung der Anzahl der Prozessoren ihr Einfluß gegenüber dem der Allreduce-Operation verringert.

Skalierbarkeit für wachsende Prozessoranzahl

Um detailliertere Aussagen über die Skalierbarkeit treffen zu können, führen wir als ersten Ansatz eine Grenzwertbetrachtung durch, bei der wir die Anzahl der Prozessoren erhöhen, indem wir p gegen unend-

lich laufen lassen und dabei s und n festhalten. Für den Speedup erhalten wir auf diese Weise:

$$\begin{aligned}\lim_{p\to\infty} S_p(n) &= \lim_{p\to\infty} \left(\frac{1}{p} + \frac{1}{pT_f(n)} T_{\text{allgather}}(p) + \frac{1}{snT_f(n)} T_{\text{allreduce}}(p) \right)^{-1} \\ &= \lim_{p\to\infty} \left(\frac{1}{pT_f(n)} T_{\text{allgather}}(p) + \frac{1}{snT_f(n)} T_{\text{allreduce}}(p) \right)^{-1} .\end{aligned} \tag{3.23}$$

Zur Berechnung des Grenzwertes müssen wir an dieser Stelle Annahmen über die Laufzeiten der Kommunikationsoperationen treffen. Da die Laufzeit der Allgather-Operation, $T_{\text{allgather}}(p)$, mit dem Faktor $1/p$ eingeht, betrachten wir dazu, wie sich $T_{\text{allgather}}(p)$ im Vergleich zu p verhält. Die in Tab. 3.1 angegebenen Beispiele für verschiedene Netzwerktopologien legen nahe, daß sich $T_{\text{allgather}}(p)$ durch $O(p)$ nach oben abschätzen läßt. Tatsächlich liefert der Algorithmus für Bäume mit $\Theta(p)$ gleichzeitig eine obere Schranke für beliebige zusammenhängende Netzwerktopologien, da für jeden zusammenhängenden Graph ein aufspannender Baum konstruiert werden kann. Somit erhalten wir:

$$\lim_{p\to\infty} S_p(n) = \lim_{p\to\infty} \left(O(1) + \frac{1}{snT_f(n)} T_{\text{allreduce}}(p) \right)^{-1} = \begin{cases} \Theta(1) & \text{für } T_{\text{allreduce}}(p) = \Theta(1), \\ 0 & \text{für } T_{\text{allreduce}}(p) = \omega(1). \end{cases} \tag{3.24}$$

Analog ergibt sich für die parallele Effizienz:

$$\begin{aligned}\lim_{p\to\infty} E_p(n) &= \lim_{p\to\infty} \left(1 + \frac{1}{T_f(n)} T_{\text{allgather}}(p) + \frac{p}{snT_f(n)} T_{\text{allreduce}}(p) \right)^{-1} \\ &= \lim_{p\to\infty} \left(1 + O(p) + \Omega(p)\right)^{-1} \\ &= 0 \ .\end{aligned} \tag{3.25}$$

Für s und n fest kann also bestenfalls ein konstanter Speedup erreicht werden, wenn die Allreduce-Operation in konstanter Zeit ausgeführt werden kann. Dies ist allerdings nur für spezielle Netzwerktopologien, z. B. einen vollständigen Graph oder ein sternförmiges Netzwerk, möglich. Für andere Netzwerktopologien, deren Durchmesser mit der Anzahl der Prozessoren wächst, steigt auch die Anzahl der zur Durchführung der Allreduce-Operation notwendigen Übertragungsschritte, was wiederum dazu führt, daß der Speedup für sehr große Prozessorzahlen gegen 0 konvergiert. Zwar ist es in der Praxis nicht möglich, die Anzahl der Prozessoren, p, über die Systemdimension, n, hinaus zu erhöhen, der theoretisch ermittelte Grenzwert für n fest und $p \to \infty$ der den Speedup beschreibenden Funktion liefert uns jedoch die praktisch relevante Aussage, daß der Speedup für festes n ab einer bestimmten Anzahl von Prozessoren nicht mehr steigt oder sogar fällt. Abhängig vom konkreten Verhältnis zwischen der Geschwindigkeit der Prozessoren und der Geschwindigkeit des Verbindungsnetzwerks des verwendeten Parallelrechners kann dies bereits für $p \leq n$ dazu führen, daß nicht alle Prozessoren des Parallelrechners effizient genutzt werden können, um ein Problem mit vorgegebener Dimension zu lösen.

Einfluß der Problemgröße auf die Skalierbarkeit

Angesichts dieser Problematik stellt sich die Frage, ob es denn wenigstens möglich ist, alle Prozessoren eines Parallelrechners effizient auszunutzen, indem man sie zur Lösung von Problemen mit hinreichend großer Dimension verwendet. Zur Beantwortung dieser Frage führen wir eine Grenzwertbetrachtung für

$n \to \infty$ und p fest durch:

$$\begin{aligned}\lim_{n\to\infty} S_p(n) &= \lim_{n\to\infty} \left(\frac{1}{p} + \frac{1}{pT_f(n)} T_{\text{allgather}}(p) + \frac{1}{snT_f(n)} T_{\text{allreduce}}(p)\right)^{-1} \\ &= \lim_{n\to\infty} \left(\frac{1}{p} + \frac{1}{pT_f(n)} T_{\text{allgather}}(p)\right)^{-1} \\ &= \begin{cases} \left(\frac{1}{p} + \frac{1}{pT_f} T_{\text{allgather}}(p)\right)^{-1} & \text{für } T_f(n) = \Theta(1) = T_f, \\ p & \text{für } T_f(n) = \omega(1), \end{cases}\end{aligned} \tag{3.26}$$

$$\begin{aligned}\lim_{n\to\infty} E_p(n) &= \lim_{n\to\infty} \left(1 + \frac{1}{T_f(n)} T_{\text{allgather}}(p) + \frac{p}{snT_f(n)} T_{\text{allreduce}}(p)\right)^{-1} \\ &= \lim_{n\to\infty} \left(1 + \frac{1}{T_f(n)} T_{\text{allgather}}(p)\right)^{-1} \\ &= \begin{cases} \left(1 + \frac{1}{T_f} T_{\text{allgather}}(p)\right)^{-1} & \text{für } T_f(n) = \Theta(1) = T_f, \\ 1 & \text{für } T_f(n) = \omega(1). \end{cases}\end{aligned} \tag{3.27}$$

Als Ergebnis erhalten wir, daß für eine beliebige, aber feste Anzahl von Prozessoren der Speedup und die Effizienz verbessert werden können, indem die Systemgröße erhöht wird. Dabei sinkt der Einfluß der Allreduce-Operation auf die Laufzeit mit wachsender Systemgröße, so daß für große n Speedup und Effizienz allein durch das Verhältnis der Laufzeit der Allgather-Operation und der Funktionsauswertungszeit bestimmt wird. Ist die Funktionsauswertungszeit unabhängig von der Systemgröße, kann ein bestimmter, von 0 verschiedener Speedup, der jedoch kleiner als p ist, nicht überschritten werden. Wächst aber die Funktionsauswertungszeit mit der Systemgröße, dominiert mit zunehmender Systemgröße die Berechnungszeit, und die Kommunikationszeit verliert an Bedeutung, so daß sich der Speedup an p und die Effizienz an 1 annähert.

Nutzung zusätzlicher Prozessoren bei wachsender Problemgröße

Versuchen wir nun, das Skalierbarkeitsverhalten für $p \to \infty$ in diesem Licht neu zu betrachten, indem wir der Frage nachgehen, ob wir bei einer Erhöhung der Systemgröße zusätzliche Prozessoren sinnvoll nutzen können, um den zunehmenden Berechnungsaufwand zu bewältigen. Dazu nehmen wir an, daß die Anzahl der Prozessoren im gleichen Verhältnis wie die Systemgröße gesteigert wird, d. h. $n/p = c = \Theta(1) \geq 1$. Dies schließt insbesondere den interessanten Fall $n = p$ ein, was bedeutet, daß zur Lösung eines Problems der Dimension n die maximal verfügbare Parallelität bezüglich des Systems ausgenutzt wird. Unter dieser Voraussetzung erhalten wir für den Speedup:

$$\begin{aligned}\lim_{n,p\to\infty} S_p(n) &= \lim_{n,p\to\infty} \left(\frac{1}{p} + \frac{1}{pT_f(n)} T_{\text{allgather}}(p) + \frac{1}{snT_f(n)} T_{\text{allreduce}}(p)\right)^{-1} \\ &= \lim_{n,p\to\infty} \left(\frac{1}{pT_f(n)} T_{\text{allgather}}(p) + \frac{1}{scpT_f(n)} T_{\text{allreduce}}(p)\right)^{-1} \\ &= \begin{cases} \Theta(1) & \text{für } T_f(n) = \Theta(1) \wedge \Big(T_{\text{allgather}}(p) = \Theta(p) \vee T_{\text{allreduce}}(p) = \Theta(p)\Big), \\ \infty & \text{sonst.} \end{cases}\end{aligned} \tag{3.28}$$

Das bedeutet, daß sich durch die proportionale Steigerung von n und p mindestens ein konstanter Speedup erhalten läßt. Die Frage ist also eindeutig mit „Ja“ zu beantworten. Unter bestimmten Voraussetzungen vergrößert sich der Speedup sogar. Dies ist der Fall, wenn die Funktionsauswertungszeit mit n wächst oder

beide Kommunikationsoperationen schneller als in $\Theta(p)$ realisiert werden können, wie dies beispielsweise bei Verwendung eines Hyperwürfel-Netzwerks der Fall ist.

Eine Verfeinerung dieser Aussage liefert die Betrachtung der parallelen Effizienz. Hierfür erhalten wir unter derselben Voraussetzung:

$$\begin{aligned}
\lim_{n,p\to\infty} E_p(n) &= \lim_{n,p\to\infty} \left(1 + \frac{1}{T_f(n)} T_{\text{allgather}}(p) + \frac{p}{snT_f(n)} T_{\text{allreduce}}(p)\right)^{-1} \\
&= \lim_{n,p\to\infty} \left(1 + \frac{1}{T_f(n)} T_{\text{allgather}}(p) + \frac{1}{scT_f(n)} T_{\text{allreduce}}(p)\right)^{-1} \\
&= \begin{cases} 1 & \text{für } T_{\text{allgather}}(p) = o(T_f(n)) \wedge T_{\text{allreduce}}(p) = o(T_f(n)), \\ \Theta(1) \in (0,1) & \text{für } T_{\text{allgather}}(p) = \Theta(T_f(n)) \wedge T_{\text{allreduce}}(p) = \Theta(T_f(n)), \\ 0 & \text{für } T_{\text{allgather}}(p) = \omega(T_f(n)) \vee T_{\text{allreduce}}(p) = \omega(T_f(n)), \end{cases}
\end{aligned} \tag{3.29}$$

Aus diesem Ergebnis wird ersichtlich, daß die Effizienz – und damit auch der genaue Verlauf der Speedup-Kurve – durch das Verhältnis von Funktionsauswertungszeit und Laufzeit der Kommunikationsoperationen bestimmt wird. So ist eine Effizienz von 1 und somit ein Speedup von p möglich, wenn die Funktionsauswertungen asymptotisch aufwendiger sind als beide Kommunikationsoperationen. Haben Funktionsauswertungen und beide Kommunikationsoperationen asymptotisch gleiche Kosten, ergibt sich im Grenzwert eine konstante Effizienz, was einem linear steigenden Speedup entspricht. Ist jedoch mindestens eine der beiden Kommunikationsoperationen teurer als die Funktionsauswertungskosten, konvergiert die Effizienz gegen 0, da in diesem Fall der Speedup langsamer wächst als die Anzahl der Prozessoren oder er sogar stagniert. Das bedeutet, daß man zwar von zusätzlichen Prozessoren profitiert, da sie die Ausführung auch für große Probleme beschleunigen, sich das Verhältnis von Laufzeitgewinn und Hardwarekosten jedoch verschlechtert.

Ein Ansteigen der parallelen Laufzeit kann aber auch bei einem gegen ∞ strebenden Speedup in der Regel nicht verhindert werden, wie die Grenzwertbetrachtung für $T_p(n)$ für $n, p \to \infty$ mit $n/p = c$ zeigt:

$$\begin{aligned}
\lim_{n,p\to\infty} T_p(n) &= \lim_{n,p\to\infty} s\frac{n}{p} T_f(n) + s\frac{n}{p} T_{\text{allgather}}(p) + T_{\text{allreduce}}(p) \\
&= \lim_{n,p\to\infty} scT_f(n) + scT_{\text{allgather}}(p) + T_{\text{allreduce}}(p) \\
&= \Theta\left(T_f(n) + T_{\text{allgather}}(p) + T_{\text{allreduce}}(p)\right) \quad .
\end{aligned} \tag{3.30}$$

Eine Vergrößerung der parallelen Laufzeit bei proportionaler Erhöhung von n und p erfolgt also nur dann nicht, wenn sowohl $T_f(n)$ als auch $T_{\text{allgather}}(p)$ und $T_{\text{allreduce}}(p)$ in konstanter Zeit ausgeführt werden können.

Zusammenfassung

Die Laufzeit einer parallelen Implementierung eines eingebetteten Runge-Kutta-Verfahrens für verteilten Adreßraum kann man als Summe des Zeitaufwands für die Berechnung der Funktionsauswertungen und der für die Ausführung der Kommunikationsoperationen benötigten Zeit abschätzen. Doch während sich der pro Prozessor erforderliche Berechnungsaufwand mit wachsender Prozessorzahl verkleinert, kann sich der Kommunikationsaufwand vergrößern. Zwar sinkt bei wachsender Prozessorzahl die Größe der mittels der Allgather-Operation ausgetauschten Nachrichten, so daß diese Operation nur einen geringen Einfluß auf das asymptotische Verhalten der Programmlaufzeit ausübt, für die Allreduce-Operation ändert sich die Nachrichtengröße jedoch nicht. Deshalb bestimmt die Anzahl der für die Ausführung der Allreduce-Operation notwendigen Übertragungsschritte das asymptotische Verhalten der Programmlaufzeit bei einer Vergrößerung der Prozessoranzahl. Infolgedessen bleibt der Speedup für eine gegebene Problemgröße ab einer bestimmten Prozessoranzahl konstant oder fällt sogar ab, was zu einer schlechten parallelen Effizienz führt.

Erhöht man jedoch unter Beibehaltung der Prozessoranzahl die Systemgröße, kann man den Speedup und die Effizienz verbessern, da der Berechnungsaufwand ansteigt, der Aufwand für die Ausführung der

Allreduce-Operation jedoch konstant bleibt, da die Größe der zu übertragenden Nachrichten unabhängig von der Problemgröße ist. Steigt die Funktionsauswertungszeit $T_f(n)$ mit der Problemgröße an, verringert sich mit wachsender Problemgröße auch der Einfluß der Allgather-Operation, so daß sich der Speedup dem optimalen Wert p annähert.

Betrachtet man die Skalierbarkeit unter dem Gesichtspunkt, daß man für die Lösung unterschiedlich großer Testprobleme einen konstanten Grad an Parallelität ausnutzen möchte, ergibt sich ein gutes Skalierbarkeitsverhalten, wenn man die Anzahl der Prozessoren im gleichen Verhältnis wie die Problemgröße erhöht. Das heißt, für große Problemgrößen und Prozessorzahlen bleibt mindestens ein konstanter Speedup erhalten, wenn man Problemgröße und Prozessoranzahl proportional erhöht. Die Entwicklung der Effizienz hängt dagegen davon ab, wie sich die Anzahl der zur Ausführung der Kommunikationsoperationen notwendigen Übertragungsschritte im Verhältnis zur Funktionsauswertungszeit steigert.

Eine genauere Analyse des Skalierbarkeitsverhaltens ist möglich, wenn die spezifische Architektur der Zielplattform bekannt ist. So kann bei bekannter Topologie des Verbindungsnetzwerks die Anzahl der Übertragungsschritte der Kommunikationsoperationen spezifiziert werden. Alternativ kann man Laufzeitformeln für die Kommunikationsoperationen durch Ausgleichsrechnung (Rauber u. Rünger 1997b) oder mittels genetischer Programmierung (Heinrich-Litan u. a. 1998) anhand empirischer Daten ermitteln (siehe auch Rauber u. Rünger 2000). Liegen Laufzeitformeln vor, die das Verhalten der Kommunikationsoperationen hinreichend genau beschreiben, kann eine detaillierte Analyse des Skalierbarkeitsverhaltens mittels Methoden der Kurvendiskussion, einschließlich der Bestimmung von Extremwerten, durchgeführt werden.

3.3.3 Einfluß des Lokalitätsverhaltens auf die Skalierbarkeit

Aus der theoretischen Analyse des Skalierbarkeitsverhaltens geht hervor, daß die Laufzeiten der Kommunikationsoperationen das asymptotische Verhalten des Speedups und der Effizienz bestimmen, da für die meisten Verbindungsnetzwerke der Kommunikationsaufwand mit steigender Prozessorzahl anwächst, während der Berechnungsaufwand pro Prozessor geringer wird. Da eine Verbesserung des Lokalitätsverhaltens die Nachrichtenübertragung über das Verbindungsnetzwerk nicht beschleunigen kann, ist deshalb zu erwarten, daß speziell für große Prozessorzahlen die Laufzeit und damit das Skalierbarkeitsverhalten durch die Kommunikationsoperationen bestimmt wird. Die nachfolgend präsentierten Laufzeitexperimente bestätigen dies und zeigen darüber hinaus, daß auf modernen Clustersystemen bereits für kleine Prozessorzahlen eine gute Skalierbarkeit durch die hohen Laufzeiten der Kommunikationsoperationen verhindert wird.

Auf eine detaillierte Analyse der Arbeitsmengen der Implementierungen für verteilten Adreßraum soll deshalb an dieser Stelle verzichtet werden. Dies geschieht auch aus dem Grund, daß eine vollständige Erfassung der Arbeitsmengen nicht ohne Einbeziehung der spezifischen Implementierung der Kommunikationsoperationen durchgeführt werden kann, da die zu versendenden Daten z. T. in verschiedene Puffer kopiert werden, wodurch sich einige Arbeitsmengen vergrößern. Jedoch sei darauf verwiesen, daß die Arbeitsmengen der Implementierungen für verteilten Adreßraum mit denen der Implementierungen für gemeinsamen Adreßraum (siehe Abschnitt 3.4.2) im wesentlichen übereinstimmen, wenn man den Einfluß der Kommunikationsoperationen außer acht läßt. Insbesondere verkleinern sich durch die Aufteilung der Iterationen der Schleifen über die Systemdimension einige Arbeitsmengen, so daß sich das Lokalitätsverhalten mit zunehmender Prozessorzahl verbessert.

3.3.4 Skalierbarkeit auf verschiedenen Rechnersystemen

Zur Untersuchung des Skalierbarkeitsverhaltens auf realen Rechnersystemen wurden Laufzeitexperimente auf mehreren Parallelrechnern mit unterschiedlicher Architektur durchgeführt.

Clustersysteme aus Standardkomponenten

Mit Hilfe von Standardkomponenten lassen sich preiswerte Parallelrechner, sogenannte Clustersysteme, aufbauen, in dem man Rechenknoten, basierend auf handelsüblicher Hardware, d. h. CPUs, Mainboards, Speicherbausteinen und Festplatten, wie sie auch in Workstations und kleineren Servern eingesetzt werden, über ein Netzwerk miteinander verbindet. Abhängig vom Budget des Käufers können solche Systeme eine große Prozessoranzahl und eine hohe theoretische Rechenleistung erreichen. Problematisch ist jedoch oft

Prozessor-anzahl	DOPRI5(4), $N = 896$, $H = 0,5$	DOPRI8(7), $N = 896$, $H = 0,5$	DOPRI5(4), $N = 384$, $H = 4,0$
1	1141	1936	316
2	2412	3321	667
4			1285

Tab. 3.2: Programmlaufzeit in Sekunden für die Implementierung (D) für verteilten Adreßraum gemessen für das Testproblem BRUSS2D-MIX auf dem Clustersystem CLiC.

die Geschwindigkeit des Verbindungsnetzwerks, da preiswerte Standardhardware zur Realisierung von Netzwerken (z. B. auf Basis von Fast Ethernet oder Gigabit Ethernet) oft eine zu geringe Übertragungsbandbreite und zu große Latenzzeiten im Verhältnis zur Arbeitsgeschwindigkeit der Prozessoren besitzen. Dieser Effekt wird oft noch durch aufwendige Softwareprotokolle verstärkt. Aber auch Verbindungsnetzwerke, die speziell für den Einsatz in Clustersystemen konzipiert wurden (z. B. Myrinet, Infiniband und Quadrics), bieten im Verhältnis zur CPU-Geschwindigkeit oft keine ausreichend hohe Geschwindigkeit für feingranulare Anwendungen.

CLiC. Der Chemnitzer Linux Cluster (CLiC), der im Jahr 2000 an der TU Chemnitz in Betrieb genommen wurde, ist ein Clustersystem, bestehend aus 528 Knoten, die durch ein Fast-Ethernet-Netzwerk mit einer Übertragungsbandbreite von 100 MBit/s verbunden sind. Die Knoten sind jeweils mit einem mit 800 MHz getakteten Pentium-III-Prozessor ausgestattet. Tabelle 3.2 zeigt einige Beispiele für Laufzeiten, die unter Verwendung von 1, 2 oder 4 Knoten für Implementierung (D) für das Testproblem BRUSS2D-MIX gemessen wurden. Dabei wurde die MPI-Bibliothek LAM/MPI 6.3.2 verwendet. Wie diese Meßergebnisse zeigen, kann auf diesem System durch die parallele Ausführung der Implementierung (D) trotz hoher Problemgröße kein Laufzeitgewinn erzielt werden. Ursache sind die sehr hohen Laufzeiten der globalen Kommunikationsoperationen, die bei einer weiteren Erhöhung der Prozessoranzahl noch vergrößert würden.

Opteron-SMP-Cluster. Heutige Clustersysteme bestehen oft aus sogenannten SMP-Knoten. *SMP* steht für englisch: *symmetric multiprocessing*. Mit diesem Begriff werden Rechnerarchitekturen mit gemeinsamem Adreßraum bezeichnet, deren Prozessoren alle die gleiche Funktionalität bieten und die gleiche Sicht auf das Rechnersystem besitzen. Zur Zeit werden bei der Anschaffung von Clustersystemen häufig SMP-Knoten mit 2 oder 4 Prozessorkernen bevorzugt, da diese momentan ein günstigeres Preis-Leistungs-Verhältnis bieten als Ein-Prozessor-Knoten. Der besondere Vorteil der Verwendung von SMP-Knoten besteht darin, daß für die Kommunikation zwischen Prozessoren innerhalb eines Knotens der im Vergleich zur Netzwerkgeschwindigkeit schnelle gemeinsame Speicher genutzt werden kann. Für die Untersuchung der parallelen Implementierungen eingebetteter Runge-Kutta-Verfahren wurde auch ein solcher SMP-Cluster genutzt. Dieses vom Lehrstuhl für Angewandte Informatik 2 der Universität Bayreuth seit 2004 betriebene System besitzt 32 Knoten mit je zwei Prozessoren vom Typ AMD Opteron DP 246 mit einer Taktfrequenz von 2 GHz. Die Knoten können über drei unterschiedliche, voneinander unabhängige Netzwerke miteinander kommunizieren. Dabei handelt es sich um ein Fast-Ethernet-Netzwerk mit 100 MBit/s, ein Gigabit-Ethernet-Netzwerk mit 1 GBit/s und ein Infiniband-Netzwerk mit 10 GBit/s. Die unterschiedlichen Geschwindigkeiten der Netzwerke erlauben es, den Einfluß der Netzwerkgeschwindigkeit auf die Laufzeit paralleler Programme zu untersuchen.

Abbildung 3.4 zeigt einige der auf diesem System für die Implementierungen (A) und (D) für das Testproblem BRUSS2D-MIX gemessenen Speedup-Werte. Zum Vergleich wurden dabei die drei unterschiedlich schnellen Verbindungsnetzwerke sowie zwei verschiedene MPI-Implementierungen herangezogen. Die Ergebnisse zeigen deutlich, daß die Programmlaufzeit durch die Laufzeit der Kommunikationsoperationen bestimmt wird. Dies ist vor allem daran erkennbar, daß sich für unterschiedliche Netzwerkgeschwindigkeiten und unterschiedliche MPI-Implementierungen der beobachtete Speedup stark verändert, während die beiden Implementierungen (A) und (D) trotz unterschiedlicher Lokalitätseigenschaften nahezu gleiche Speedups erreichen. Im Vergleich zum älteren CLiC-System verbessert sich die Skalierbarkeit auf diesem System nur geringfügig. Zwar kann für beide MPI-Implementierungen und für alle Verbindungsnetzwerke ein Speedup größer als 1 erreicht werden, wenn nur zwei Prozesse auf ein und demselben Knoten gestar-

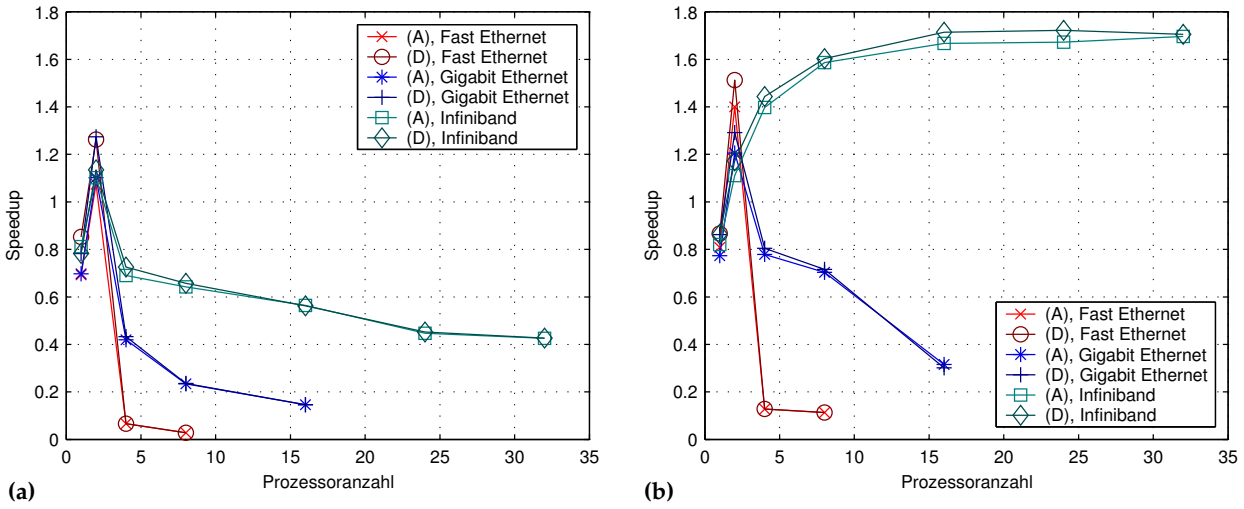


Abb. 3.4: Speedups der Implementierungen (A) und (D) für verteilten Adreßraum gemessen für das Testproblem BRUSS2D-MIX mit dem Verfahren DOPRI5(4), $N = 1000$ und $H = 4,0$ auf einem Opteron-SMP-Cluster für zwei verschiedene MPI-Implementierungen: **(a)** LAM/MPI 7.1.1, **(b)** MPICH 1.2.5.

tet werden, sobald jedoch Kommunikation zwischen mehreren Knoten erforderlich ist, verschlechtert sich die Laufzeit in den meisten Fällen. Bei Verwendung von LAM/MPI 7.1.1 konnte für keines der drei Verbindungsnetzwerke eine Laufzeitverbesserung durch die Benutzung von zwei oder mehr Knoten erreicht werden. Nur für MPICH 1.2.5 konnte eine Laufzeitverbesserung durch die Verwendung mehrerer Knoten bei Verwendung des Infiniband-Netzwerkes beobachtet werden. Allerdings wächst auch in diesem Fall der Speedup nur bis zu einem Wert von ca. 1.7 bei Verwendung von 16 Knoten an und stagniert danach.

Cray T3E

Bessere Speedups können auf speziell entworfenen, eng gekoppelten Parallelrechnern erzielt werden, deren Verbindungsnetzwerk schnell und mit geringem Overhead arbeitet und auf die verwendeten Prozessoren abgestimmt ist. Beispielsweise wurden auf einem Parallelrechner vom Typ Cray T3E-1200, der von 1997 bis 2004 am Forschungszentrum Jülich betrieben wurde, in den durchgeführten Experimenten mit der Implementierung (D) und dem Testproblem BRUSS2D-MIX Speedups bis zu 17,7 gemessen (siehe Abb. 3.5). Die Cray T3E war mit 512 Prozessoren vom Typ DEC Alpha 21164 mit einer Taktfrequenz von 600 MHz ausgestattet, die durch ein 3D-Torus-Netzwerk mit einer Übertragungsgeschwindigkeit von 500 MB/s verbunden waren. Der Speicher war physikalisch verteilt organisiert, wurde jedoch zu einem globalen Adreßraum zusammengefaßt. Der Datenaustausch zwischen den Prozessoren erfolgte durch Remote-Speicherzugriffe mittels spezieller Operationen über sogenannte E-Register.

Im Vergleich zu den zuvor betrachteten Clustersystemen und teilweise auch im Vergleich zu moderneren Parallelrechnern mit gemeinsamem Speicher erreichte die untersuchte Implementierung (D) auf der Cray T3E die beste Skalierbarkeit. Der Grund dafür ist das mit 500 MB/s im Vergleich zum Prozessortakt von 600 MHz sehr schnelle Verbindungsnetzwerk. Zum Vergleich: Das Infiniband-Netzwerk des Opteron-Clusters erreicht eine Übertragungsgeschwindigkeit von ca. 800 MB/s und ist damit nur ungefähr 1,6-mal schneller. Die Opteron-Prozessoren sind mit 2 GHz jedoch $3^1/_3$-mal höher getaktet als die Alpha-Prozessoren der T3E und können pro Takt mehr Daten verarbeiten. Dennoch ist es nicht möglich, alle Prozessoren der T3E effizient auszunutzen. Denn abhängig von der Problemgröße und der Stufenzahl sinkt der Speedup, wenn eine bestimmte Anzahl von Prozessoren überschritten wird. In den durchgeführten Experimenten konnten ca. 32 bis 48 Prozessoren effizient genutzt werden. Der höchste Speedup für BRUSS2D-MIX von 17,7 wurde mit $N = 896$ und $H = 0,5$ für das 13-stufige Verfahren DOPRI8(7) unter Verwendung von 32 Prozessoren gemessen, was einer parallelen Effizienz von 0,55 entspricht. In Experimenten mit geringerer Problemgröße und/oder geringerer Stufenzahl wurde der maximale Speedup ebenfalls für 32 Prozessoren erreicht, er betrug jedoch nur 12,2 bzw. 14,2.

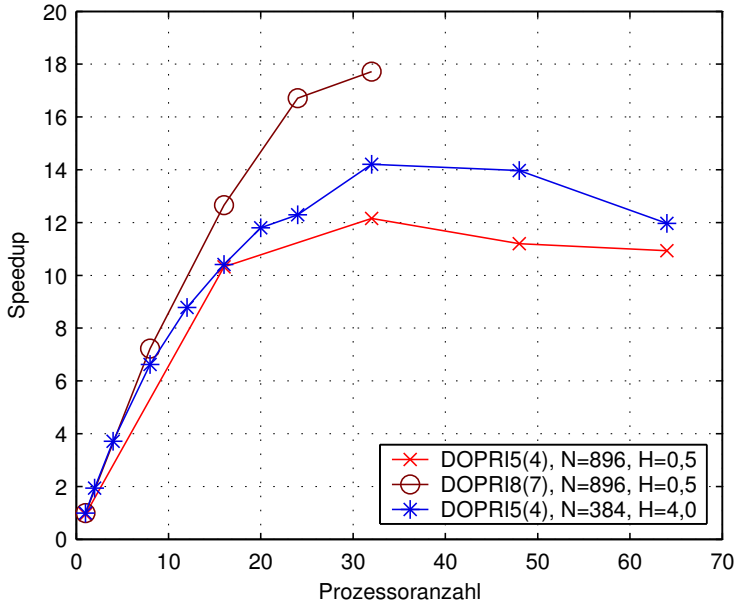


Abb. 3.5: Speedups der Implementierung (D) für verteilten Adreßraum gemessen für das Testproblem BRUSS2D-MIX auf einer Cray T3E-1200.

Parallelrechner mit gemeinsamem Adreßraum

Auch auf Parallelrechnern mit gemeinsamem Adreßraum kann MPI zur Kommunikation zwischen Prozessen genutzt werden. Für solche Systeme optimierte MPI-Implementierungen können den gemeinsamen physikalischen Adreßraum zum Nachrichtenaustausch zwischen den Prozessen effizient ausnutzen, indem die Nachrichten direkt in den virtuellen Adreßraum des empfangenden Prozesses kopiert werden, ohne daß Schichten eines Netzwerkprotokolls durchlaufen oder spezielle Remote-Speicherzugriffsoperationen auf Softwareebene ausgeführt werden müssen.

Sun Fire 6800. Eine Untersuchung des Skalierbarkeitsverhaltens der eingebetteten Runge-Kutta-Verfahren wurde deshalb auch auf zwei SMP-Systemen, u. a. auf einer Sun Fire 6800, die seit 2001 an der Martin-Luther-Universität Halle-Wittenberg betrieben wird, durchgeführt. Diese Maschine ist mit 24 UltraSPARC-III-Cu-Prozessoren mit einer Taktfrequenz von 900 MHz und einem 8 MB großen L2-Cache ausgestattet. Aufgrund einer hohen Nachfrage nach Rechenzeit auf dieser Maschine durch andere Nutzer konnten zur Durchführung der Experimente jedoch nur bis zu 8 Prozessoren genutzt werden.

Die für die Implementierungen (A) und (D) für das Testproblem BRUSS2D-MIX mit dem Verfahren DOPRI5(4) gemessenen Speedup-Werte sind in Abb. 3.6 dargestellt. Für die betrachteten Prozessorzahlen wurden ähnlich gute Speedup-Werte wie auf der Cray T3E gemessen. Im Unterschied zur T3E spielen hier jedoch die Lokalitätseigenschaften eine größere Rolle. Für $N = 250$ beträgt der Speicherbedarf pro Vektor 977 KB. Das bedeutet, daß 8 vollständige Vektoren im 8 MB großen L2-Cache zwischengespeichert werden können. Da sich die Arbeitsmengen bei Erhöhung der Prozessorzahl durch die Aufteilung der Vektoren verkleinern, kann bereits bei Verwendung von zwei Prozessoren die gesamte Arbeitsmenge eines Zeitschrittes im L2-Cache zwischengespeichert werden. Daher ist auch mehr als eine Verdopplung des Speedups von 0,86 auf 1,82 für Implementierung (A) und von 0,87 auf 1,77 für Implementierung (D) möglich. Jedoch verhindert der hohe Overhead, der durch die Ausführung der Kommunikationsoperationen entsteht, daß der Speedup, bezogen auf die Laufzeit der schnellsten sequentiellen Implementierung, den Idealwert erreicht. Beim Übergang von zwei auf vier Prozessoren ist dagegen keine Verdopplung des Speedups mehr möglich, da sich bereits bei Verwendung von zwei Prozessoren alle innerhalb eines Zeitschrittes zugegriffenen Daten im Cache befinden, der Kommunikationsoverhead durch die Verwendung zusätzlicher Prozessoren jedoch steigt. Bei weiterer Erhöhung der Prozessorzahl sinkt die Effizienz weiter bis auf einen Wert von 0,58 bei

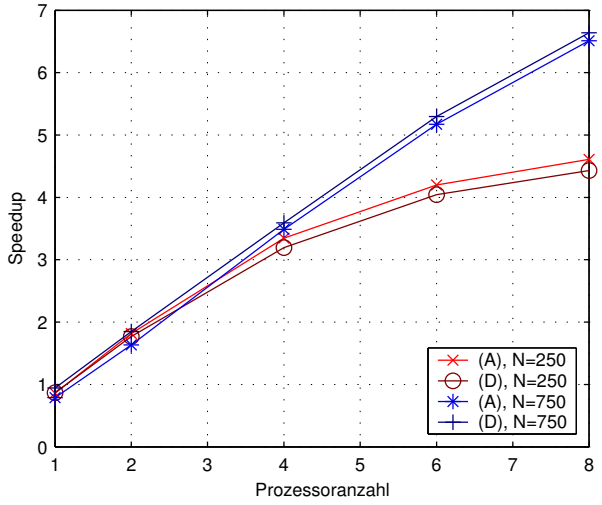


Abb. 3.6: Speedups der Implementierungen (A) und (D) für verteilten Adreßraum gemessen für das Testproblem BRUSS2D-MIX mit dem Verfahren DOPRI5(4) und $H = 4,0$ auf einer Sun Fire 6800.

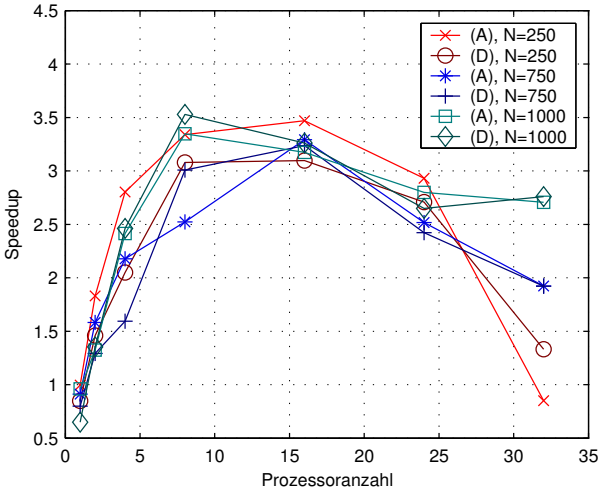


Abb. 3.7: Speedups der Implementierungen (A) und (D) für verteilten Adreßraum gemessen für das Testproblem BRUSS2D-MIX mit dem Verfahren DOPRI5(4) und $H = 4,0$ auf einem Knoten des Supercomputers JUMP.

einem Speedup von 4,61 für Implementierung (A) und bis auf einen Wert von 0,55 bei einem Speedup von 4,43 für Implementierung (D) bei Verwendung von 8 Prozessoren. Implementierung (A) erreicht aufgrund der besseren Ausnutzung der räumlichen Lokalität bereits ab zwei Prozessoren eine bessere Laufzeit als Implementierung (D) und baut diesen Vorsprung mit zunehmender Prozessorzahl aus.

Für $N = 750$ belegt ein vollständiger Vektor rund 8,6 MB. Infolgedessen kann die Arbeitsmenge eines Zeitschrittes auch bei einer Aufteilung auf 8 Prozessoren nicht vollständig im L2-Cache gespeichert werden. Der Anteil der im Cache zwischengespeicherten Daten an der Arbeitsmenge eines Zeitschrittes nimmt jedoch mit steigender Prozessorzahl zu. Durch diese Verbesserung der Speicherzugriffslokalität wird der mit wachsender Prozessorzahl steigende Kommunikationsoverhead kompensiert, woraus sich ein etwa linear steigender Verlauf der Speedup-Kurve für die betrachteten Prozessorzahlen ergibt. Für 8 Prozessoren wurde ein Speedup von 6,51 für Implementierung (A) und ein Speedup von 6,64 für Implementierung (D) gemessen, was einer Effizienz von 0,81 bzw. 0,83 entspricht. Bei genauerer Betrachtung der Effizienz stellt man fest, daß diese für Implementierung (D), die für alle Prozessorzahlen eine geringere Laufzeit erreicht als Implementierung (A), mit wachsender Prozessorzahl generell geringfügig abfällt, während sie für Implementierung (A) bis zu einer Anzahl von vier Prozessoren ansteigt und erst danach abfällt. Dieser Effekt wird durch das Lokalitätsverhalten der Implementierungen hervorgerufen, da aufgrund der Tatsache, daß nicht die gesamte Arbeitsmenge eines Zeitschrittes im Cache gespeichert werden kann, die zeitliche Lokalität von großer Bedeutung ist, sie mit zunehmender Prozessoranzahl und sinkender Größe der Arbeitsmengen jedoch gegenüber der räumlichen Lokalität an Einfluß verliert.

JUMP. Der „Jülicher Multiprozessor“ (JUMP) wird seit 2003 am Forschungszentrum Jülich betrieben und war nach Installation der letzten Ausbaustufe Anfang 2004 für einige Zeit Europas schnellster Supercomputer. Das Gesamtsystem besteht aus 41 IBM-p690-Servern mit je 32 Power4+-Prozessorkernen mit einer Taktfrequenz von 1,7 GHz. Innerhalb der SMP-Knoten können die Prozessoren über einen gemeinsamen Adreßraum kommunizieren. Die Verbindung zwischen den Knoten wird über einen sogenannten High-Performance-Switch hergestellt. Bei der Untersuchung der parallelen Implementierungen eingebetteter Runge-Kutta-Verfahren zeigte sich bereits bei Verwendung eines einzelnen Knotens, wobei die gestarteten Prozesse über den schnellen gemeinsamen Speicher kommunizieren konnten, daß keine gute Skalierbarkeit auf diesem System erreicht werden kann (siehe Abb. 3.7). Die Untersuchung wurde deshalb nicht auf mehrere Knoten ausgedehnt, da die dann notwendige Kommunikation über das langsamere externe Verbindungsnetzwerk zu einer weiteren Verschlechterung von Laufzeit und Speedup führen würde.

Abbildung 3.7 zeigt Speedup-Werte, die für die Implementierungen (A) und (D) bei Verwendung des Testproblems BRUSS2D-MIX für die Problemgrößen $N = 250$, $N = 750$ und $N = 1000$ gemessen wurden. Für die kleinste untersuchte Problemgröße, $N = 250$, erreichen beide Implementierungen, (A) und (D), ihren maximalen Speedup von 3,47 bzw. 3,10 bei Verwendung von 16 Prozessoren. Bei Steigerung der Prozessorzahl bis auf 32 fällt der Speedup auf 0,85 bzw. 1,33 ab. Implementierung (A) ist bei Verwendung von $N = 250$ für Prozessorzahlen bis zu 24 erkennbar schneller als Implementierung (D). Nur wenn 32 Prozessoren benutzt werden, erreicht Implementierung (D) einen höheren Speedup. Für größere Probleme, d. h. für $N = 750$ und $N = 1000$, sind die Unterschiede zwischen den beiden Implementierungen weniger deutlich und liegen im Bereich des auf dieser Maschine auftretenden Meßfehlers. Für $N = 750$ beträgt der maximale Speedup der Implementierungen (A) und (D) 3,29 bzw. 3,24. Er wird wie für $N = 250$ bei Verwendung von 16 Prozessoren erreicht. Bei Verwendung zusätzlicher Prozessoren fällt der Speedup im Vergleich zu $N = 250$ langsamer ab, und beide Implementierungen erreichen für 32 Prozessoren einen Speedup von 1,92. Für $N = 1000$ erreichen beide Implementierungen ihren maximalen Speedup von 3,35 bzw. 3,53 bereits bei Verwendung von 8 Prozessoren. Im Vergleich zu $N = 750$ fällt der Speedup bei einer Vergrößerung der Prozessorzahl wiederum langsamer ab. Werden 32 Prozessoren verwendet, erreicht Implementierung (A) für $N = 1000$ einen Speedup von 2,71 und Implementierung (D) einen Speedup von 2,76.

Insgesamt betrachtet, werden für keine der untersuchten Problemgrößen so hohe Speedup-Werte erreicht, wie dies auf der Sun Fire 6800 der Fall war. Da sich die Laufzeiten der beiden Implementierungen insbesondere für Probleme mit großer Systemdimension aufgrund der relativ geringen Differenz und der wechselnden Reihenfolge nicht eindeutig vergleichen lassen, ist anzunehmen, daß die Laufzeiten wesentlich durch die Ausführung der Kommunikationsoperationen bestimmt werden und daß der Einfluß der Lokalität mit wachsender Systemgröße sinkt. Dafür spricht auch die Tatsache, daß sich die Skalierbarkeit durch Erhöhung der Problemgröße nicht wesentlich verbessert.

3.3.5 Zusammenfassung

Parallelrechner mit verteiltem Adreßraum sind weit verbreitet, da sich diese Speicherarchitektur sowohl zur Realisierung preiswerter Clustersysteme aus Standardkomponenten als auch zum Aufbau der weltweit schnellsten Supercomputer eignet. Aufgrund des verteilten Adreßraumes erfolgt die Kommunikation zwischen den Prozessoren eines solchen Rechnersystems durch den Austausch von Nachrichten. Eine standardisierte Programmierschnittstelle für diese Art der Kommunikation ist z. B. MPI.

Die Ausnutzung der Parallelität bezüglich des Differentialgleichungssystems bei der Implementierung eingebetteter Runge-Kutta-Verfahren erfordert in jeder Stufe einen Austausch des Argumentvektors mit Hilfe einer Multibroadcastoperation, wie z. B. `MPI_Allgather()`. Zur Bestimmung des globalen Maximums des Fehlervektors kann eine spezielle Form einer Multiakkumulationsoperation benutzt werden, wie sie `MPI_Allreduce()` realisiert. Die Verwendung einer blockweisen Verteilung ermöglicht eine effiziente Übergabe der zu übertragenden Nachrichten an bzw. durch die Kommunikationsoperationen, ohne daß eine zusätzliche Vor- oder Nachbereitung der Daten erforderlich ist.

Eine theoretische Betrachtung des Skalierbarkeitsverhaltens zeigt, daß zwar die pro Prozessor erforderliche Berechnungszeit mit wachsender Prozessorzahl sinkt, der Kommunikationsaufwand jedoch unter bestimmten Bedingungen anwachsen kann. Asymptotisch wird der Speedup für wachsende Prozessorzahl durch die Laufzeit der Allreduce-Operation dominiert, da die Nachrichtengröße für die Allgather-Operationen mit wachsender Prozessorzahl geringer wird. Steigt also die Laufzeit der Allreduce-Operation mit wachsender Prozessorzahl an, wie dies für die meisten typischen Verbindungsnetzwerke der Fall ist, strebt der Speedup asymptotisch gegen 0, was auf ein schlechtes Skalierbarkeitsverhalten hinweist. Die theoretische Analyse zeigt jedoch auch, daß die Skalierbarkeit durch Erhöhung der Problemgröße verbessert werden kann. Dies gilt insbesondere dann, wenn die Kosten für die Funktionsauswertung einer Komponente des Differentialgleichungssystems mit steigender Problemgröße zunehmen, da in diesem Fall der Einfluß der Kommunikationskosten im Verhältnis zu den Berechnungskosten mit steigender Problemgröße sinkt. Weiterhin sagt die theoretische Skalierbarkeitsanalyse eine gute Skalierbarkeit für den Fall voraus, daß Prozessoranzahl und Problemgröße in gleichem Verhältnis gesteigert werden. Dies entspricht der Situation, daß versucht wird, den aus einer Erhöhung der Problemgröße resultierenden zusätzlichen Parallelitätsgrad durch Verwendung zusätzlicher Prozessoren auszunutzen. Eine besonders gute Skalierbarkeit ergibt

sich in dieser Situation vor allem dann, wenn die Kommunikationsoperationen asymptotisch nicht teurer sind als die Funktionsauswertungskosten.

Das Lokalitätsverhalten der parallelen Implementierungen verbessert sich mit zunehmender Prozessorzahl, da durch die Aufteilung der zu verarbeitenden Daten die Größe einiger Arbeitsmengen abnimmt. Dies hat zur Folge, daß die Berechnungszeit schneller als mit dem Reziproken der Prozessorzahl sinkt. Da die Laufzeit für große Prozessorzahlen jedoch durch die Kommunikationskosten dominiert wird, hat das Lokalitätsverhalten in der Regel lediglich für kleine Prozessorzahlen Einfluß auf die Skalierbarkeit.

In Laufzeitexperimenten auf mehreren unterschiedlichen parallelen Rechnersystemen konnten meist nur geringe Speedups erzielt werden. Speziell auf aus Standardkomponenten aufgebauten Clustersystemen ist es z. T. schwierig, auch nur eine geringe Beschleunigung durch die parallele Ausführung zu erreichen, da die Kommunikationskosten aufgrund des im Verhältnis zur Prozessorgeschwindigkeit oft langsamen Netzwerks enorm hoch im Vergleich zur Berechnungszeit sind. Doch auch auf Systemen mit gemeinsamem Adreßraum, auf denen ein effizienter Nachrichtenaustausch zwischen Prozessen über den gemeinsamen Adreßraum realisiert werden kann, setzt eine gute Skalierbarkeit voraus, daß bei wachsender Prozessoranzahl die Verbesserung der Lokalitätseigenschaften die steigenden Kommunikationskosten aufwiegen. Die beste Skalierbarkeit wurde auf einer Cray T3E-1200 beobachtet. Bei diesem System handelt es sich um einen für die Kommunikation mittels Nachrichtenaustausch optimierten Supercomputer, der ein im Vergleich zur Prozessorgeschwindigkeit wesentlich schnelleres Verbindungsnetzwerk besitzt als die betrachteten Clustersysteme.

3.4 Parallelrechner mit gemeinsamem Adreßraum

Viele moderne parallele Rechnersysteme mit gemeinsamem Adreßraum sind SMP-Systeme (von englisch: *symmetric multiprocessing*), deren Prozessoren alle die gleiche Funktionalität bieten und die gleiche Sicht auf das Rechnersystem besitzen. In früheren SMP-Systemen, die in der Regel nur wenige Prozessoren besaßen, tauschten alle Prozessoren und alle Speichermodule Daten über einen gemeinsamen Bus aus. Speicherzugriffe führten daher unabhängig von der zugegriffenen Adresse und dem zugreifende Prozessor zu annähernd gleichen Latenzzeiten (englisch: *uniform memory access*, UMA). Mit zunehmender Prozessorzahl wächst jedoch die zwischen Prozessoren und Speicher pro Zeiteinheit zu transferierende Datenmenge, da alle Prozessoren pro Zeiteinheit insgesamt mehr Daten verarbeiten können. Das Bussystem wirkt in diesem Fall wie ein Flaschenhals, den alle zu verarbeitenden Daten passieren müssen und der deshalb die Skalierbarkeit begrenzt. Moderne Systeme akzeptieren daher ungleichförmige Speicherzugriffszeiten (englisch: *non-uniform memory access*, NUMA), um eine größere Anzahl von Prozessoren effizient einsetzen zu können. Beispiele sind der IBM eServer p5 595 mit bis zu 64 Prozessorkernen, die Sun Fire E25K mit bis zu 144 Prozessorkernen sowie die SGI Altix 4700, deren Architektur einen gemeinsamen Adreßraum für mehrere tausend Prozessoren zur Verfügung stellen kann.

Auf NUMA-System können Speicherzugriffe auf entfernten Speicher vorkommen, der sich unter Umständen in einem anderen Rack als der zugreifende Prozessor befindet, das in der gegenüberliegenden Ecke einer Maschinenhalle steht. Die den Zugriff realisierenden Signale müssen daher mehrere Meter Kabel durchlaufen, sowie eine Hierachie von Switches bzw. Routern, welche die Knoten des Verbindungsnetzwerks bilden. Entsprechend hoch können die Latenzzeiten für derartige Speicherzugriffe ausfallen. Um solch hohe Speicherzugriffszeiten zu „kaschieren“, verwenden derartige Systeme oft eine tiefe Hierarchie von Caches mit hoher Kapazität. Ein gutes Lokalitätsverhalten ist daher für ein paralleles Programm die Voraussetzung, um eine gute Skalierbarkeit erreichen zu können.

Wie auf Rechnern mit verteiltem Adreßraum ist auch im Fall eines gemeinsamen Adreßraumes eine Synchronisation der beteiligten Kontrollflüsse erforderlich. Da die Kontrollflüsse aufgrund des gemeinsamen Adreßraumes auf die selben Datenstrukturen zugreifen können, bezeichnet man das zugehörige Programmiermodell als *Programmierung mit gemeinsamen Variablen*. Die Kontrollflüsse selbst werden als *Threads* bezeichnet. Explizite Kommunikationsoperationen zum Austausch von Nachrichten sind nicht erforderlich, jedoch müssen zeitkritische Abläufe beim Zugriff auf gemeinsam genutzte Daten vermieden werden. Die Basis dafür bieten atomare Maschineninstruktionen wie z. B. Fetch & Add und Compare & Swap, die zur Implementierung komplexerer Synchronisationsmechanismen genutzt werden können. Dem Anwendungsprogrammierer werden diese Synchronisationsmechanismen durch Bibliotheken mit einer standar-

disierten, plattformunabhängigen Schnittstelle zur Verfügung gestellt. Beispiele für solche Standardschittstellen sind OpenMP (OpenMP API 2.5 2005) und POSIX Threads (Butenhof 1997).

3.4.1 Implementierung für gemeinsamen Adreßraum

Bei der Implementierung für verteilten Adreßraum war es erforderlich, den in der jeweiligen Stufe benötigten Argumentvektor zu replizieren, d. h. für jeden beteiligten Prozeß eine vollständige Kopie des Argumentvektors herzustellen, damit durch die Funktionsauswertung auf beliebige Elemente des Argumentvektors lesend zugegriffen werden kann. Aufgrund des verteilten Adreßraumes war dazu ein Nachrichtenaustausch notwendig. Bei der Implementierung für einen gemeinsamen Adreßraum kann prinzipiell auf die gleiche Weise vorgegangen werden, indem jeder Thread eigene Datenstrukturen verwendet. An die Stelle des Nachrichtenaustauschs über das Verbindungsnetzwerk tritt in diesem Fall ein einfaches Kopieren der benötigten Datenblöcke innerhalb des gemeinsamen Adreßraumes. Durch geeignete Synchronisationsmechanismen muß dabei sichergestellt werden, daß Argumentvektorelemente erst dann kopiert werden, wenn ihre Berechnung abgeschlossen ist. Durch das Vorhalten mehrerer Kopien erhöht sich jedoch der Speicherplatzbedarf, und das Kopieren der Daten erfordert Rechenzeit, die nicht direkt zum Fortschritt der Berechnung beiträgt und somit einen Synchronisationsoverhead darstellt.

Ein gemeinsamer Adreßraum bietet jedoch die Möglichkeit, gemeinsam durch mehrere Threads genutzte Datenstrukturen zu realisieren. Bezüglich der Argumentvektoren bedeutet dies, daß nur eine einzige Kopie des jeweiligen Argumentvektors benötigt wird, auf die alle Threads zur Durchführung der Funktionsauswertungen der ihnen zugeordneten Komponenten lesend zugreifen können. Es muß lediglich sichergestellt werden, daß erst dann mit der Durchführung der Funktionsauswertungen begonnen wird, wenn die Berechnung aller Elemente des betreffenden Argumentvektors abgeschlossen ist. Dazu kann eine sogenannte *Barrier-Operation* verwendet werden. Diese Operation setzt erst dann die Ausführung des aufrufenden Threads fort, wenn alle anderen beteiligten Threads ebenfalls diese Operation aufgerufen haben. Falls, wie in den Implementierungen (A) und (E), nicht alle Argumentvektoren gespeichert werden, sondern der Speicherbereich für den Argumentvektor einer Stufe durch die Daten des Argumentvektors der nachfolgenden Stufe überschrieben wird, muß zusätzlich sichergestellt werden, daß alle Funktionsauswertungen abgeschlossen sind, bevor die dafür benötigten Daten überschrieben werden. Dazu kann wiederum eine Barrier-Operation verwendet werden. Alternativ besteht die Möglichkeit, einen zweiten temporären Speicherbereich zur Verfügung zu stellen und die beiden Speicherbereiche abwechselnd durch Vertauschen der Anfangszeiger lesend für die Funktionsauswertungen bzw. schreibend für die Berechnung des nachfolgenden Argumentvektors zu nutzen.

Der Vorteil der Nutzung gemeinsamer Datenstrukturen besteht vor allem darin, daß weniger Speicherplatz benötigt wird und ein aufwendiges Kopieren von Daten entfällt. Stattdessen werden durch die Funktionsauswertungen genau die Werte aus einer gemeinsamen Datenstruktur gelesen, die tatsächlich benötigt werden. Eine Implementierung auf Basis verteilter Datenstrukturen müßte ohne Kenntnis des speziellen Zugriffsmusters des zu integrierenden Problems alle Elemente des Argumentvektors replizieren. Auf NUMA-Systemen kann die Verwendung verteilter Datenstrukturen jedoch Vorteile mit sich bringen, sofern die von einem Thread benutzten Datenstrukturen in einem lokalen Speicher des Prozessors abgelegt werden können, auf dem dieser Thread ausgeführt wird. Da auf diesen Speicher schneller zugegriffen werden kann als auf entfernten Speicher eines anderen Prozessors, kann die resultierende Verringerung der Speicherzugriffszeiten den Overhead des Kopiervorganges unter Umständen aufwiegen. Diese Möglichkeit besteht insbesondere dann, wenn durch die Funktionsauswertungen Argumentvektorkomponenten mehrfach benutzt werden und das Zugriffsverhalten der Funktionsauswertungen zu häufigen Cache-Fehlzugriffen führt, die Zugriffe auf den Hauptspeicher erforderlich machen. Die Voraussetzung dafür ist, daß der Thread während seiner gesamten Laufzeit auf ein und demselben Prozessor ausgeführt wird, d. h., daß er an einen Prozessor *gebunden* wird.

Das Binden von Threads an Prozessoren hat generell den Vorteil, daß Situationen vermieden werden, in denen von einem Thread in den Cache eines Prozessors geladene Daten aufgrund einer Migration des Threads auf einen anderen Prozessor nicht mehr wiederverwendet werden können. Um ein gutes Lokalitätsverhalten zu ermöglichen, ist es weiterhin erforderlich, eine günstige Datenverteilung zu wählen, welche die Vektorkomponenten derart auf die beteiligten Threads bzw. Prozessoren verteilt, daß im Cache befindliche Daten möglichst häufig wiederverwendet werden können. Die höchste räumliche Lokalität

wird erreicht, wenn den Threads zusammenhängende Vektorblöcke zugeordnet werden, d. h. eine blockweise Datenverteilung verwendet wird. Für viele gewöhnliche Differentialgleichungssysteme, deren Funktionsauswertungen nur Komponenten des Argumentvektors innerhalb einer bestimmten Zugriffsdistanz verwenden, führt eine blockweise Datenverteilung darüber hinaus zu einer guten zeitlichen Lokalität.

Zu beachten ist bei der Festlegung der Datenverteilung, daß auf Parallelrechnern mit gemeinsamem Adreßraum mehrere Kopien eines Datums in den Caches verschiedener Prozessoren existieren können. Man spricht hierbei vom sogenannten *Sharing* von Cachezeilen. Dabei bezeichnet man als *True Sharing* den Fall, daß mehrere Caches eine Kopie der gleichen Cachezeile beinhalten, weil die betreffenden Prozessoren auf die selben Datenelemente innerhalb der Cachezeile zugreifen. Im Gegensatz dazu spricht man von *False Sharing*, wenn das Vorhandensein der mehrfachen Kopien einer Cachezeile in verschiedenen Caches dadurch hervorherufen wurde, daß die betreffenden Prozessoren auf unterschiedliche Datenelemente innerhalb der Cachezeile zugegriffen haben. Um Cache-Fehlzugriffe aufgrund von Sharing-Effekten zu vermeiden, sollten die Grenzen der blockweisen Datenverteilung nach Möglichkeit so festgelegt werden, daß die Datenelemente einer Cachezeile dem selben Thread zugeordnet werden. Dazu ist es gegebenenfalls erforderlich, die Vektoren so im Speicher anzuordnen, daß die erste Adresse eines Vektors auf das erste Element einer Cachezeile abgebildet wird. Falls die Vektorelemente einen Speicherplatzbedarf von mehreren Maschinenworten besitzen, muß in diesem Zusammenhang auch dafür Sorge getragen werden, daß alle Bestandteile eines Vektorelements auf die gleiche Cachezeile abgebildet werden.

Zur Durchführung der Schrittweitenkontrolle muß das globale Maximum ϵ der lokal von den einzelnen Threads bestimmten Maximalwerte ϵ_{local} der Komponenten des Fehlervektors ermittlet werden. Die Implementierungen für verteilten Adreßraum greifen dazu auf die MPI-Operation `MPI_Allreduce()` zurück und verlassen sich darauf, daß die jeweilige MPI-Bibliothek eine effiziente Implementierung dieser Funktion bereitstellt. Bei einer Implementierung für gemeinsamen Adreßraum kann bei Verwendung von OpenMP zu diesem Zweck eine `reduction`-Klausel eingesetzt werden. Für POSIX Threads ist dagegen keine derartige Reduktionsoperation spezifiziert; sie läßt sich jedoch auf verschiedene Arten nachbilden. Für große Prozessorzahlen kann es dabei lohnenswert sein, einen hierarchischen Algorithmus mit logarithmischer Komplexität in der Anzahl der Prozessoren einzusetzen, da so Rechenoperationen eingespart werden können und unter asymptotischen Gesichtspunkten die bestmögliche Skalierbarkeit erwartet werden kann. In der Praxis hat man es jedoch in der Regel mit einer verhältnismäßig kleinen Prozessoranzahl zu tun. Aktuelle SMP-Systeme besitzen beispielsweise meist weniger als 100 Prozessoren. Eine sequentielle Bestimmung des Maximums ist daher oft die effizienteste Strategie, da sie mit nur wenigen Synchronisationsoperationen auskommt.

Im folgenden betrachten wir parallele Versionen der Implementierungen (A), (E), (D) und (Dblock) für gemeinsamen Adreßraum, die in den Abb. 3.8 bis 3.11 dargestellt sind. Alle vorgestellten Implementierungen basieren auf POSIX Threads und verwenden jeweils nur eine einzige, gemeinsam genutzte Kopie der Stufen- und Argumentvektoren. Die Datenverteilung erfolgt blockweise, ohne dabei die Zuordnung der Datenelemente zu Cachezeilen zu beachten. Zur Trennung der Argumentvektorberechnung und der Funktionsauswertungen wird in jeder Stufe eine Barrier-Operation ausgeführt. Die Implementierungen (A) und (E) verwenden nur einen einzigen temporären Argumentvektor und führen deshalb in jeder Stufe eine zweite Barrier-Operation aus. Die Bestimmung des globalen Maximums ϵ innerhalb der Schrittweitenkontrolle erfolgt sequentiell durch den Thread mit der ID 0. Zuvor wird eine Barrier-Operation ausgeführt, um sicherzustellen, daß die Berechnung der lokalen Maximalwerte ϵ_{local} abgeschlossen wurde. Anschließend wird ebenfalls eine Barrier-Operation ausgeführt, damit die übrigen Threads nicht mit dem Wert ϵ vor dessen endgültiger Berechnung weiterrechnen. Eine mögliche Verbesserung bestünde darin, eine spezielle Reduktionsoperation zu implementieren, die ähnlich einer Barrier-Operation arbeitet, jedoch mit dem Unterschied, daß eintreffende Threads den bis zu diesem Zeitpunkt bestimmten globalen Maximalwert mit ihrem lokalen Maximalwert vergleichen und gegebenenfalls aktualisieren.

3.4.2 Theroretische Analyse des Lokalitätsverhaltens

Auf Parallelrechnern mit gemeinsamem Adreßraum wird die Skalierbarkeit zwar ebenfalls durch Synchronisationsoperationen beeinflußt, für speicherintensive Programme ist es jedoch mindestens ebenso wichtig, wie schnell Speicherzugriffsoperationen ausgeführt werden können. Eingebettete Runge-Kutta-Verfahren benötigen zur Synchronisation nur wenige Barrier-Operationen, führen aber zur Berechnung und Verarbei-

```
1:  // Berechne Stufenvektor v_1
2:  for (j := j_first; j ≤ j_last; j++) v_1[j] := f_j(t + c_1 h, η);
3:  // Berechne Stufenvektoren v_2, ..., v_s
4:  for (l := 2; l ≤ s; l++)
5:  {
6:      barrier();
7:      for (j := j_first; j ≤ j_last; j++) w[j] := a_l1 v_1[j];
8:      for (i := 2; i < l; i++)
9:          for (j := j_first; j ≤ j_last; j++) w[j] += a_li v_i[j];
10:     for (j := j_first; j ≤ j_last; j++) w[j] := η[j] + h w[j];
11:     barrier();
12:     for (j := j_first; j ≤ j_last; j++) v_l[j] := f_j(t + c_l h, w);
13: }
14: // Berechne Δη und Fehlervektor e
15: for (j := j_first; j ≤ j_last; j++)
16: {
17:     Δη[j] := b_1 v_1[j];
18:     e[j] := b̃_1 v_1[j];
19: }
20: for (l := 2; l ≤ s; l++)
21:     for (j := j_first; j ≤ j_last; j++)
22:     {
23:         Δη[j] += b_l v_l[j];
24:         e[j] += b̃_l v_l[j];
25:     }
26: // Berechne Skalierungsfaktoren
27: for (j := j_first; j ≤ j_last; j++) s[j] := |η[j]| + |h · v_1[j]|;
```

Abb. 3.8: Parallele Version der Implementierung (A) für gemeinsamen Adreßraum.

```
1:  // Berechne Stufenvektor v_1 und Skalierungsfaktoren

2:  for (j := j_first; j ≤ j_last; j++)
3:  {
4:      v_1[j] := f_j(t + c_1 h, η);
5:      s[j] := |η[j]| + |h v_1[j]|;
6:  }

7:  // Berechne Stufenvektoren v_2, ..., v_s

8:  for (l := 2; l ≤ s; l++)
9:  {
10:     for (j := j_first; j ≤ j_last; j++)
11:     {
12:         w := a_{l1} v_1[j];
13:         for (i := 2; i < l; i++) w += a_{li} v_i[j];
14:         w[j] := η[j] + hw;
15:     }

16:     barrier();

17:     for (j := j_first; j ≤ j_last; j++) v_l[j] := f_j(t + c_l h, w);

18:     barrier();
19: }

20: // Berechne Δη und Fehlervektor e

21: for (j := j_first; j ≤ j_last; j++)
22: {
23:     d := b_1 v_1[j];
24:     e := b̃_1 v_1[j];

25:     for (l := 2; l ≤ s; l++)
26:     {
27:         d += b_l v_l[j];
28:         e += b̃_l v_l[j];
29:     }

30:     Δη[j] := d;
31:     e[j] := e;
32: }
```

Abb. 3.9: Parallele Version der Implementierung (E) für gemeinsamen Adreßraum.

```
1:  // Berechne Stufenvektor v_1

2:  barrier();

3:  for (j := j_first; j ≤ j_last; j++)
4:  {
5:      v~ := h f_j(t + c_1 h, η);
6:      s[j] := |η[j]| + |v~|;

7:      for (i := 2; i ≤ s; i++) w_i[j] := η[j] + a_i1 v~;

8:      Δη[j] := b_1 v~;
9:      e[j] := b~_1 v~;
10: }

11: // Berechne Stufenvektoren v_2, ..., v_s

12: for (l := 2; l ≤ s; l++)
13: {
14:     barrier();

15:     for (j := j_first; j ≤ j_last; j++)
16:     {
17:         v~ := h f_j(t + c_l h, w_l);

18:         for (i := l + 1; i ≤ s; i++) w_i[j] += a_il v~;

19:         Δη[j] += b_l v~;
20:         e[j] += b~_l v~;
21:     }
22: }
```

Abb. 3.10: Parallele Version der Implementierung (D) für gemeinsamen Adreßraum.

tung von Elementen der verschiedenen Vektoren eine Vielzahl von Speicherzugriffsoperationen aus. Diese müssen unter Umständen mehrere Bussysteme und Schaltelemente (Switches oder Router) durchlaufen, wodurch sich Engpässe ausbilden können, die die Ausführung der Speicherzugriffsoperationen verzögern und somit die Skalierbarkeit begrenzen. Ein gutes Lokalitätsverhalten, daß es ermöglicht, den größten Teil der Speicherzugriffsoperationen aus den vorhandenen Caches zu beantworten und somit das Verbindungssystem zu den Speichermodulen zu entlasten, ist deshalb die Voraussetzung für das Erreichen einer guten Skalierbarkeit. Aus diesem Grund widmet sich dieser Abschnitt der theoretischen Analyse des Lokalitätsverhaltens der im vorigen Abschnitt vorgestellten Implementierungen, um eine Einschätzung und einen Vergleich des Lokalitätsverhaltens der Implementierungen und ihres Potentials für eine gute Skalierbarkeit zu ermöglichen.

Bezüglich des Speicherplatzbedarfs gibt es für die im vorigen Abschnitt vorgestellten parallelen Implementierungsvarianten keine Unterschiede zu den entsprechenden sequentiellen Ausgangsimplementierungen (vgl. Tab. 2.2). Es werden die gleichen Vektoren verwendet wie in den Ausgangsimplementierungen, jedoch werden diese von allen Threads gemeinsam genutzt. Durch diese Aufteilung ergeben sich im Vergleich zu den sequentiellen Implementierungen kleinere Arbeitsräume. Eine Aufstellung der Arbeitsräume der parallelen Implementierungen für gemeinsamen Adreßraum unter Annahme der Verwendung von p Threads bzw. Prozessoren zeigen Tab. 3.3 und Tab. 3.4.

Die Analyse der Arbeitsräume der in den parallelen Implementierungen enthaltenen Schleifen ergibt, daß sich durch die Parallelisierung ausschließlich die Arbeitsräume der Schleifen über die Systemdimen-

Zeile	I	$\lvert I\rvert$	R_I	R_L	$\lvert R_I\rvert$	$\lvert R_L\rvert$
Implementierung (A)						
2	$j = j_{\text{first}}, \ldots, j_{\text{last}}$	$\lceil n/p \rceil$	$\mathbf{v}_1[j]$; D Komp. von η	$\lceil n/p \rceil$ Komp. von $\mathbf{v}_1$; $\lceil n/p \rceil + D - 1$ Komp. von η	$D+1$	$2\lceil n/p \rceil + D - 1$
4	$l = 2, \ldots, s$	$s-1$	$\lceil n/p \rceil$ Komp. von $\mathbf{v}_1, \ldots, \mathbf{v}_l, \eta$; $\lceil n/p \rceil + D - 1$ Komp. von $\mathbf{w}$	$\lceil n/p \rceil$ Komp. von $\mathbf{v}_1, \ldots, \mathbf{v}_s, \eta$; $\lceil n/p \rceil + D - 1$ Komp. von $\mathbf{w}$	$(l+2)\lceil n/p \rceil + D - 1$	$(s+2)\lceil n/p \rceil + D - 1$
7	$j = j_{\text{first}}, \ldots, j_{\text{last}}$	$\lceil n/p \rceil$	$\mathbf{w}[j]$, $\mathbf{v}_1[j]$	$\lceil n/p \rceil$ Komp. von $\mathbf{w}$, $\mathbf{v}_1$	2	$2\lceil n/p \rceil$
8	$i = 2, \ldots, l-1$	$l-2$	$\lceil n/p \rceil$ Komp. von $\mathbf{w}$, $\mathbf{v}_i$	$\lceil n/p \rceil$ Komp. von $\mathbf{w}$, $\mathbf{v}_2, \ldots, \mathbf{v}_{l-1}$	$2\lceil n/p \rceil$	$(l-1)\lceil n/p \rceil$
9	$j = j_{\text{first}}, \ldots, j_{\text{last}}$	$\lceil n/p \rceil$	$\mathbf{w}[j]$, $\mathbf{v}_i[j]$	$\lceil n/p \rceil$ Komp. von $\mathbf{w}$, $\mathbf{v}_i$	2	$2\lceil n/p \rceil$
10	$j = j_{\text{first}}, \ldots, j_{\text{last}}$	$\lceil n/p \rceil$	$\mathbf{w}[j]$, $\eta[j]$	$\lceil n/p \rceil$ Komp. von $\mathbf{w}$, η	2	$2\lceil n/p \rceil$
12	$j = j_{\text{first}}, \ldots, j_{\text{last}}$	$\lceil n/p \rceil$	$\mathbf{v}_l[j]$; D Komp. von $\mathbf{w}$	$\lceil n/p \rceil + D - 1$ Komp. von $\mathbf{w}$; $\lceil n/p \rceil$ Komp. von $\mathbf{v}_l$	$D+1$	$2\lceil n/p \rceil + D - 1$
15	$j = j_{\text{first}}, \ldots, j_{\text{last}}$	$\lceil n/p \rceil$	$\mathbf{v}_1[j]$, $\Delta\eta[j]$, $\mathbf{e}[j]$	$\lceil n/p \rceil$ Komp. von $\mathbf{v}_1$, $\Delta\eta$, $\mathbf{e}$	3	$3\lceil n/p \rceil$
20	$l = 2, \ldots, s$	$s-1$	$\lceil n/p \rceil$ Komp. von $\mathbf{v}_l$, $\Delta\eta$, $\mathbf{e}$	$\lceil n/p \rceil$ Komp. von $\mathbf{v}_2, \ldots, \mathbf{v}_s$, $\Delta\eta$, $\mathbf{e}$	$3\lceil n/p \rceil$	$(s+1)\lceil n/p \rceil$
21	$j = j_{\text{first}}, \ldots, j_{\text{last}}$	$\lceil n/p \rceil$	$\mathbf{v}_l[j]$, $\Delta\eta[j]$, $\mathbf{e}[j]$	$\lceil n/p \rceil$ Komp. von $\mathbf{v}_l$, $\Delta\eta$, $\mathbf{e}$	3	$3\lceil n/p \rceil$
27	$j = j_{\text{first}}, \ldots, j_{\text{last}}$	$\lceil n/p \rceil$	$\mathbf{v}_1[j]$, $\mathbf{s}[j]$, $\eta[j]$	$\lceil n/p \rceil$ Komp. von $\mathbf{v}_1$, $\mathbf{s}$, η	3	$3\lceil n/p \rceil$
Implementierung (E)						
2	$j = j_{\text{first}}, \ldots, j_{\text{last}}$	$\lceil n/p \rceil$	$\mathbf{v}_1[j]$, $\mathbf{s}[j]$; D Komp. von η	$\lceil n/p \rceil$ Komp. von $\mathbf{v}_1$, $\mathbf{s}$; $\lceil n/p \rceil + D - 1$ Komp. von η	$D+2$	$3\lceil n/p \rceil + D - 1$
8	$l = 2, \ldots, s$	$s-1$	$\lceil n/p \rceil$ Komp. von $\mathbf{v}_1, \ldots, \mathbf{v}_l, \eta$; $\lceil n/p \rceil + D - 1$ Komp. von $\mathbf{w}$	$\lceil n/p \rceil$ Komp. von $\mathbf{v}_1, \ldots, \mathbf{v}_s, \eta$; $\lceil n/p \rceil + D - 1$ Komp. von $\mathbf{w}$	$(l+2)\lceil n/p \rceil + D - 1$	$(s+2)\lceil n/p \rceil + D - 1$
10	$j = j_{\text{first}}, \ldots, j_{\text{last}}$	$\lceil n/p \rceil$	$\mathbf{w}[j]$, $\mathbf{v}_1[j], \ldots, \mathbf{v}_{l-1}[j]$, $\eta[j]$	$\lceil n/p \rceil$ Komp. von $\mathbf{w}$, $\mathbf{v}_1, \ldots, \mathbf{v}_{l-1}$, η	$l+1$	$(l+1)\lceil n/p \rceil$
13	$i = 2, \ldots, l-1$	$l-2$	$\mathbf{v}_i[j]$	$\mathbf{v}_2[j], \ldots, \mathbf{v}_{l-1}[j]$	1	$l-2$
17	$j = j_{\text{first}}, \ldots, j_{\text{last}}$	$\lceil n/p \rceil$	$\mathbf{v}_l[j]$; D Komp. von $\mathbf{w}$	$\lceil n/p \rceil + D - 1$ Komp. von $\mathbf{w}$; $\lceil n/p \rceil$ Komp. von $\mathbf{v}_l$	$D+1$	$2\lceil n/p \rceil + D - 1$
21	$j = j_{\text{first}}, \ldots, j_{\text{last}}$	$\lceil n/p \rceil$	$\mathbf{v}_1[j], \ldots, \mathbf{v}_s[j]$, $\Delta\eta[j]$, $\mathbf{e}[j]$	$\lceil n/p \rceil$ Komp. von $\mathbf{v}_1, \ldots, \mathbf{v}_s$, $\Delta\eta$, $\mathbf{e}$	$s+2$	$(s+2)\lceil n/p \rceil$
25	$l = 2, \ldots, s$	$s-1$	$\mathbf{v}_l[j]$	$\mathbf{v}_2[j], \ldots, \mathbf{v}_s[j]$	1	$s-1$

Tab. 3.3: Arbeitsräume der in den allgemeinen parallelen Implementierungen verwendeten Schleifen.

Zeile	I	$\|I\|$	R_I	R_L	$\|R_I\|$	$\|R_L\|$
Implementierung (D)						
3	$j = j_{\text{first}}, \ldots, j_{\text{last}}$	$\lceil n/p \rceil$	$\mathbf{w}_2[j], \ldots, \mathbf{w}_s[j], \Delta\eta[j], \mathbf{e}[j], \mathbf{s}[j]$; D Komp. von η	$\lceil n/p \rceil + D - 1$ Komp. von η; $\lceil n/p \rceil$ Komp. von $\mathbf{w}_2, \ldots, \mathbf{w}_s, \Delta\eta, \mathbf{e}, \mathbf{s}$	$D + s + 2$	$(s+3)\lceil n/p \rceil + D - 1$
7	$i = 2, \ldots, s$	$s - 1$	$\mathbf{w}_i[j], \eta[j]$	$\mathbf{w}_2[j], \ldots, \mathbf{w}_s[j], \eta[j]$	2	s
12	$l = 2, \ldots, s$	$s - 1$	$\lceil n/p \rceil$ Komp. von $\mathbf{w}_{l+1}, \ldots, \mathbf{w}_s, \Delta\eta, \mathbf{e}$; $\lceil n/p \rceil + D - 1$ Komp. von $\mathbf{w}_l$	$\lceil n/p \rceil$ Komp. von $\Delta\eta, \mathbf{e}$; $\lceil n/p \rceil + D - 1$ Komp. von $\mathbf{w}_2, \ldots, \mathbf{w}_s$	$(s-l+3)\lceil n/p \rceil + D - 1$	$(s+1)\lceil n/p \rceil + (s-1)(D-1)$
15	$j = j_{\text{first}}, \ldots, j_{\text{last}}$	$\lceil n/p \rceil$	$\mathbf{w}_{l+1}[j], \ldots, \mathbf{w}_s[j], \Delta\eta[j], \mathbf{e}[j]$; D Komp. von $\mathbf{w}_l$	$\lceil n/p \rceil$ Komp. von $\mathbf{w}_{l+1}, \ldots, \mathbf{w}_s, \Delta\eta, \mathbf{e}$; $\lceil n/p \rceil + D - 1$ Komp. von $\mathbf{w}_l$	$D + s - l + 2$	$(s-l+3)\lceil n/p \rceil + D - 1$
18	$i = l+1, \ldots, s$	$s - l$	$\mathbf{w}_i[j]$	$\mathbf{w}_{l+1}[j], \ldots, \mathbf{w}_s[j]$	1	$s - l$
Implementierung (Dblock)						
3	$j = j_{\text{first}}, j_{\text{first}} + B, \ldots, j_{\text{last}}$	$\lceil \lceil n/p \rceil / B \rceil$	$\hat{\mathbf{v}}$; B Komp. von $\mathbf{w}_2, \ldots, \mathbf{w}_s, \Delta\eta, \mathbf{e}, \mathbf{s}$; $B + D - 1$ Komp. von η	$\hat{\mathbf{v}}$; $\lceil n/p \rceil + D - 1$ Komp. von η; $\lceil n/p \rceil$ Komp. von $\mathbf{w}_2, \ldots, \mathbf{w}_s, \Delta\eta, \mathbf{e}, \mathbf{s}$	$D + (s+4)B - 1$	$(s+3)\lceil n/p \rceil + D - 1 + B$
5	$k = 0, \ldots, B-1$	B	$\hat{\mathbf{v}}[k+1]$; D Komp. von η	$\hat{\mathbf{v}}$; $B + D - 1$ Komp. von η	$D + 1$	$2B + D - 1$
6	$k = 0, \ldots, B-1$	B	$\hat{\mathbf{v}}[k+1], \eta[j+k], \mathbf{s}[j+k]$	$\hat{\mathbf{v}}$; B Komp. von $\eta, \mathbf{s}$	3	$3B$
7	$i = 2, \ldots, s$	$s - 1$	$\hat{\mathbf{v}}$; B Komp. von $\mathbf{w}_i, \eta$	$\hat{\mathbf{v}}$; B Komp. von $\mathbf{w}_2, \ldots, \mathbf{w}_s, \eta$	$3B$	$(s+1)B$
8	$k = 0, \ldots, B-1$	B	$\hat{\mathbf{v}}[k+1], \mathbf{w}_i[j+k], \eta[j+k]$	$\hat{\mathbf{v}}$; B Komp. von $\mathbf{w}_i, \eta$	3	$3B$
9	$k = 0, \ldots, B-1$	B	$\hat{\mathbf{v}}[k+1], \Delta\eta[j+k]$	$\hat{\mathbf{v}}$; B Komp. von $\Delta\eta$	2	$2B$
10	$k = 0, \ldots, B-1$	B	$\hat{\mathbf{v}}[k+1], \mathbf{e}[j+k]$	$\hat{\mathbf{v}}$; B Komp. von $\mathbf{e}$	2	$2B$
13	$l = 2, \ldots, s$	$s - 1$	$\hat{\mathbf{v}}$; $\lceil n/p \rceil + D - 1$ Komp. von $\mathbf{w}_l$; $\lceil n/p \rceil$ Komp. von $\mathbf{w}_{l+1}, \ldots, \mathbf{w}_s, \Delta\eta, \mathbf{e}$	$\hat{\mathbf{v}}$; $\lceil n/p \rceil$ Komp. von $\Delta\eta, \mathbf{e}$; $\lceil n/p \rceil + D - 1$ Komp. von $\mathbf{w}_2, \ldots, \mathbf{w}_s$	$(s-l+3)\lceil n/p \rceil + D - 1 + B$	$(s+1)\lceil n/p \rceil + (s-1)(D-1) + B$
16	$j = j_{\text{first}}, j_{\text{first}} + B, \ldots, j_{\text{last}}$	$\lceil \lceil n/p \rceil / B \rceil$	$B + D - 1$ Komp. von $\mathbf{w}_l$; $\hat{\mathbf{v}}$; B Komp. von $\mathbf{w}_{l+1}, \ldots, \mathbf{w}_s, \Delta\eta, \mathbf{e}$	$\hat{\mathbf{v}}$; $\lceil n/p \rceil + D - 1$ Komp. von $\mathbf{w}_l$; $\lceil n/p \rceil$ Komp. von $\mathbf{w}_{l+1}, \ldots, \mathbf{w}_s, \Delta\eta, \mathbf{e}$	$D + (s-l+4)B - 1$	$(s-l+3)\lceil n/p \rceil + D - 1 + B$
18	$k = 0, \ldots, B-1$	B	$\hat{\mathbf{v}}[k+1]$; D Komp. von $\mathbf{w}_l$	$\hat{\mathbf{v}}$; $B + D - 1$ Komp. von $\mathbf{w}_l$	$D + 1$	$2B + D - 1$
19	$i = l+1, \ldots, s$	$s - l$	$\hat{\mathbf{v}}$; B Komp. von $\mathbf{w}_i$	$\hat{\mathbf{v}}$; B Komp. von $\mathbf{w}_{l+1}, \ldots, \mathbf{w}_s$	$2B$	$(s-l+1)B$
20	$k = 0, \ldots, B-1$	B	$\hat{\mathbf{v}}[k+1], \mathbf{w}_i[j+k]$	$\hat{\mathbf{v}}$; B Komp. von $\mathbf{w}_i$	2	$2B$
21	$k = 0, \ldots, B-1$	B	$\hat{\mathbf{v}}[k+1], \Delta\eta[j+k]$	$\hat{\mathbf{v}}$; B Komp. von $\Delta\eta$	2	$2B$
22	$k = 0, \ldots, B-1$	B	$\hat{\mathbf{v}}[k+1], \mathbf{e}[j+k]$	$\hat{\mathbf{v}}$; B Komp. von $\mathbf{e}$	2	$2B$

Tab. 3.4: Arbeitsräume der in den allgemeinen parallelen Implementierungen verwendeten Schleifen (Fortsetzung).

```
1:  // Berechne Stufenvektor v_1
2:  barrier();
3:  for (j := j_first; j ≤ j_last; j += B)
4:  {
5:      for (k := 0; k < min(B, j_last − j + 1); k++) v̂[k + 1] := h f_{j+k}(t + c_1 h, η);
6:      for (k := 0; k < min(B, j_last − j + 1); k++) s[j + k] := |η[j + k]| + |v̂[k + 1]|;
7:      for (i := 2; i ≤ s; i++)
8:          for (k := 0; k < min(B, j_last − j + 1); k++) w_i[j + k] := η[j + k] + a_{i1} v̂[k + 1];
9:      for (k := 0; k < min(B, j_last − j + 1); k++) Δη[j + k] := b_1 v̂[k + 1];
10:     for (k := 0; k < min(B, j_last − j + 1); k++) e[j + k] := b̃_1 v̂[k + 1];
11: }
12: // Berechne Stufenvektoren v_2, ..., v_s
13: for (l := 2; l ≤ s; l++)
14: {
15:     barrier();
16:     for (j := j_first; j ≤ j_last; j += B)
17:     {
18:         for (k := 0; k < min(B, j_last − j + 1); k++) v̂[k + 1] := h f_{j+k}(t + c_l h, w_l);
19:         for (i := l + 1; i ≤ s; i++)
20:             for (k := 0; k < min(B, j_last − j + 1); k++) w_i[j + k] += a_{il} v̂[k + 1];
21:         for (k := 0; k < min(B, j_last − j + 1); k++) Δη[j + k] += b_l v̂[k + 1];
22:         for (k := 0; k < min(B, j_last − j + 1); k++) e[j + k] += b̃_l v̂[k + 1];
23:     }
24: }
```

Abb. 3.11: Parallele Version der Implementierung (Dblock) für gemeinsamen Adreßraum.

sion und – als Konsequenz davon – auch die Arbeitsräume der sie umhüllenden Schleifen verändern. Die Arbeitsräume der innersten Schleifen der Implementierungen (D) und (E), welche über Stufen laufen und die zeitliche Lokalität durch Wiederverwendung einer skalaren Variable ausnutzen, verändern sich dagegen nicht. Ebenso hat die Parallelisierung keinen Einfluß auf die Arbeitsräume der Tile-Schleifen innerhalb der Implementierung (Dblock). Die Veränderung der Arbeitsräume der Schleifen über die Systemdimension hängt davon ab, ob in der jeweiligen Schleife eine Funktionsauswertung erfolgt. Durch die blockweise Aufteilung der Iterationen führt jeder Thread maximal $\lceil n/p \rceil$ Iterationen der Schleifen über die Systemdimension aus. Für die Vektorzugriffe außerhalb von Funktionsauswertungen bedeutet dies, daß jetzt anstelle aller n Vektorelemente nur noch maximal $\lceil n/p \rceil$ Vektorelemente verwendet werden. Der Arbeitsraum dieser Schleifen verkleinert sich also mit dem Faktor $1/p$ gegenüber der sequentiellen Implementierung. Der Arbeitsraum von Schleifen über die Systemdimension, in denen Funktionsauswertungen durchgeführt werden, verringert sich dagegen nicht in gleichem Maße. Zwar werden auch hier durch die blockweise Aufteilung der Iterationen nur Funktionsauswertungen für $\lceil n/p \rceil$ Komponenten ausgeführt, diese können jedoch auf beliebige Komponenten des jeweiligen Argumentvektors zugreifen. Im ungünstigsten Fall werden für eine Funktionsauswertung sogar alle Komponenten des Argumentvektors benötigt, so daß dieser Argumentvektor vollständig in den Arbeitsraum der Schleife eingeht. Zur Beschreibung des Arbeitsraumes

dieser Schleifen unter Berücksichtigung der spezifischen Eigenschaften des gewöhnlichen Differentialgleichungssystems kann die Konstante $D = 2d(\mathbf{f}) + 1$ verwendet werden, die, unter Zuhilfenahme der Zugriffsdistanz der Funktion $\mathbf{f}$, eine obere Schranke für die Anzahl der während einer Funktionsauswertung benutzten Komponenten des Argumentvektors angibt. Die Gesamtzahl der während der Auswertung eines Blocks von $\lceil n/p \rceil$ Komponenten zugegriffenen Argumentvektorelemente läßt sich somit als $\lceil n/p \rceil + D - 1$ angeben.

Insgesamt läßt sich festhalten, daß durch die Parallelisierung eine Reihe kleinerer Arbeitsräume innerster Schleifen, deren Grenzen unabhängig von der Systemdimension sind, unverändert bleiben. Die großen Arbeitsräume der Schleifen über die Systemdimension verkleinern sich jedoch mit einem Faktor von $1/p$, sofern sie keine Funktionsauswertungen enthalten. Ist D ein im Vergleich zu $\lceil n/p \rceil$ kleiner Wert, verkleinern sich auch die Arbeitsräume der Schleifen über die Systemdimension, in denen Funktionsauswertungen stattfinden, annähernd mit einem Faktor von $1/p$. Da sich die Arbeitsräume mit wachsender Prozessorzahl verkleinern, kann man allgemein von einem sehr guten Lokalitätsverhalten sprechen, das theoretisch sogar superlineare Speedups ermöglicht. Dies ist dann der Fall, wenn wichtige Arbeitsmengen bei einer sequentiellen Ausführung nicht in den Cache passen, bei einer parallelen Ausführung durch die Aufteilung der Vektoren jedoch kleinere Arbeitsmengen entstehen, die vom Cache aufgenommen werden können. Besitzt das zu integrierende Problem eine große Zugriffsdistanz, hat dies einen entscheidenden Einfluß auf das Lokalitätsverhalten. Besonders betroffen hiervon sind die Implementierungen (D) und (Dblock), die alle s Argumentvektoren getrennt speichern. In diesem Fall erhöht die Funktionsauswertung jedes Argumentvektors den Gesamtarbeitsraum eines Zeitschrittes um $D - 1$. Die Implementierungen (A) und (E) führen Funktionsauswertungen dagegen nur für den Approximationsvektor η sowie für einen temporären Argumentvektor $\mathbf{w}$ durch. Der Arbeitsraum eines Zeitschrittes vergrößert sich deshalb durch die Funktionsauswertung nur um $2(D - 1)$. Für Probleme mit großer Zugriffsdistanz könnte dies zu einer besseren Skalierbarkeit der Implementierungen (A) und (E) im Vergleich zu den Implementierungen (D) und (Dblock) führen.

3.4.3 Vergleich des Laufzeitverhaltens

Zur Untersuchung des Laufzeitverhaltens der parallelen Implementierungen für gemeinsamen Adreßraum wurden Laufzeitexperimente auf einem Itanium-2-SMP sowie auf einem Knoten des Supercomputers JUMP durchgeführt.

Auswirkung der Arbeitsmengen auf die Laufzeit

In Fortsetzung der Experimente zur Untersuchung der Auswirkung der Arbeitsmengen der sequentiellen Implementierungen auf deren Laufzeit wurden gleichartige Experimente auch für die parallelen Implementierungen für gemeinsamen Adreßraum durchgeführt. Dazu wurde das selbe Zielsystem, ein Itanium-2-SMP vom Typ HP Integrity rx5670 verwendet, das in Anhang A näher beschrieben wird. Als Compiler wurde wiederum der Intel C/C++ Compiler 9.0 verwendet. Um die Auswirkungen der Arbeitsmengen beobachten zu können, betrachten wir, wie bereits zuvor für die sequentiellen Implementierungen, die normierte Laufzeit, d. h. die Laufzeit pro Zeitschritt und pro Systemkomponente, in Abhängigkeit von der Systemgröße. Abbildung 3.12 zeigt die Ergebnisse für BRUSS2D-MIX und BRUSS2D-ROW mit dem Verfahren DOPRI5(4). Weitere Ergebnisse für MEDAKZO mit dem Verfahren DOPRI8(7) sind in Abb. 3.13 dargestellt.

Bei Betrachtung der normierten Laufzeitkurven fällt zunächst auf, daß sich bei Ausführung mit nur einem Thread die Laufzeiten der Implementierungen und insbesondere auch die Reihenfolge der Implementierungen im Vergleich zu den sequentiellen Implementierungsvarianten stark verändert haben. Als Hauptursache dafür wurden Interferenzen zwischen Daten und Instruktionen identifiziert, die durch die Verschiebung der relativen Codepositionen u. a. aufgrund des Einfügens der Barrier-Operationen hervorgerufen wurden. Weiterhin ist auffällig, daß der bereits für die sequentiellen Implementierungen beobachtete Sprung bei einer Systemgröße von 16 384, der durch ein verändertes verhalten der Funktion `malloc()` ab einer Blockgröße von 128 KB hervorgerufen wird, ebenfalls bei Verwendung der parallelen Implementierungen auftritt. Hier ist allerdings hauptsächlich nur die Implementierung (D) betroffen, während der Effekt für andere Implementierungen kaum wahrnehmbar ist. Die Erhöhung der normierten Laufzeit der

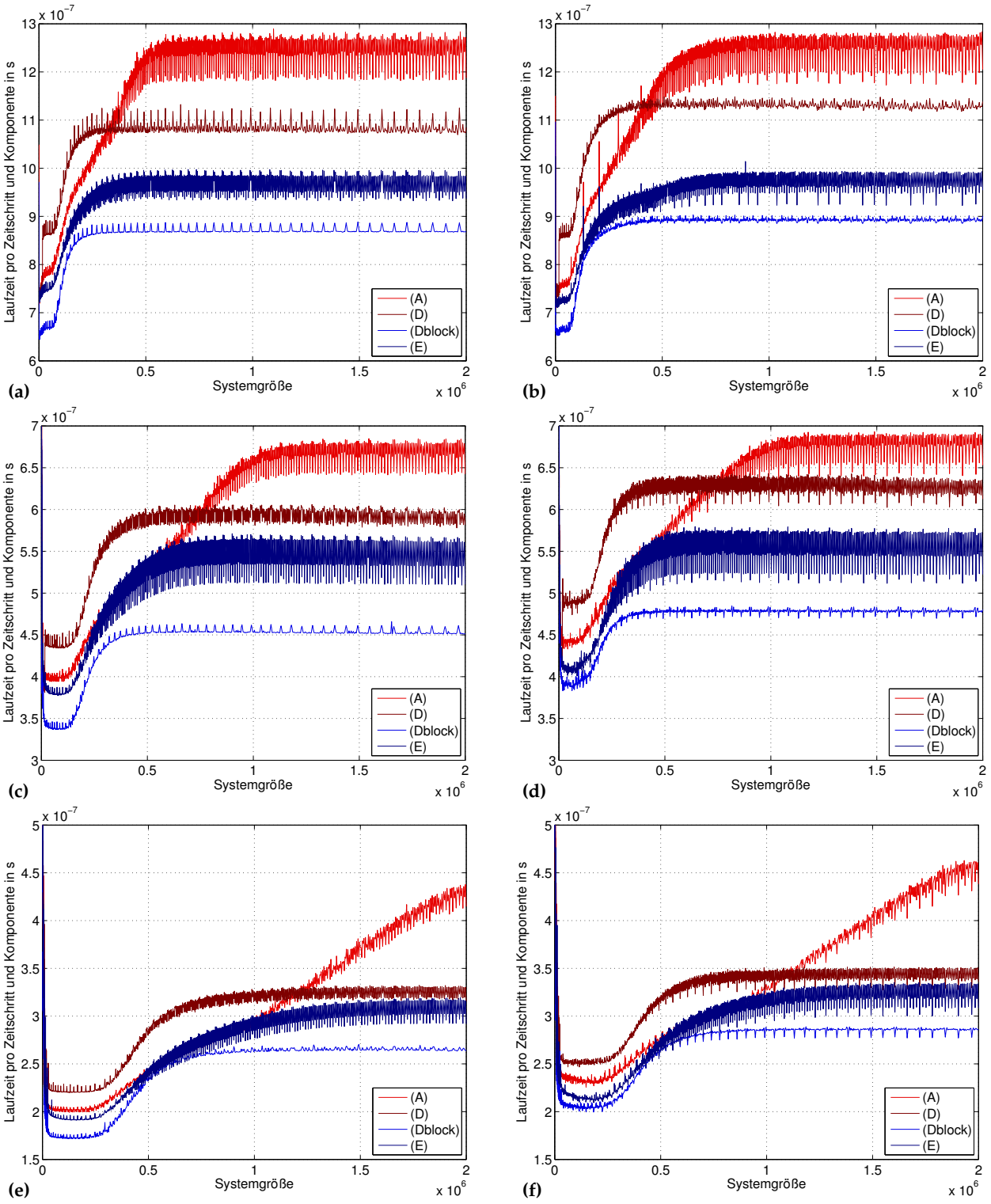


Abb. 3.12: Laufzeit der allgemeinen parallelen Implementierungen für gemeinsamen Adreßraum pro Zeitschritt und Komponente in Abhängigkeit von der Systemgröße auf einem Itanium-2-System. **(a)** BRUSS2D-MIX, 1 Thread, **(b)** BRUSS2D-ROW, 1 Thread, **(c)** BRUSS2D-MIX, 2 Threads, **(d)** BRUSS2D-ROW, 2 Threads, **(e)** BRUSS2D-MIX, 4 Threads und **(f)** BRUSS2D-ROW, 4 Threads. Für beide Testprobleme wurde das Verfahren DOPRI5(4) verwendet.

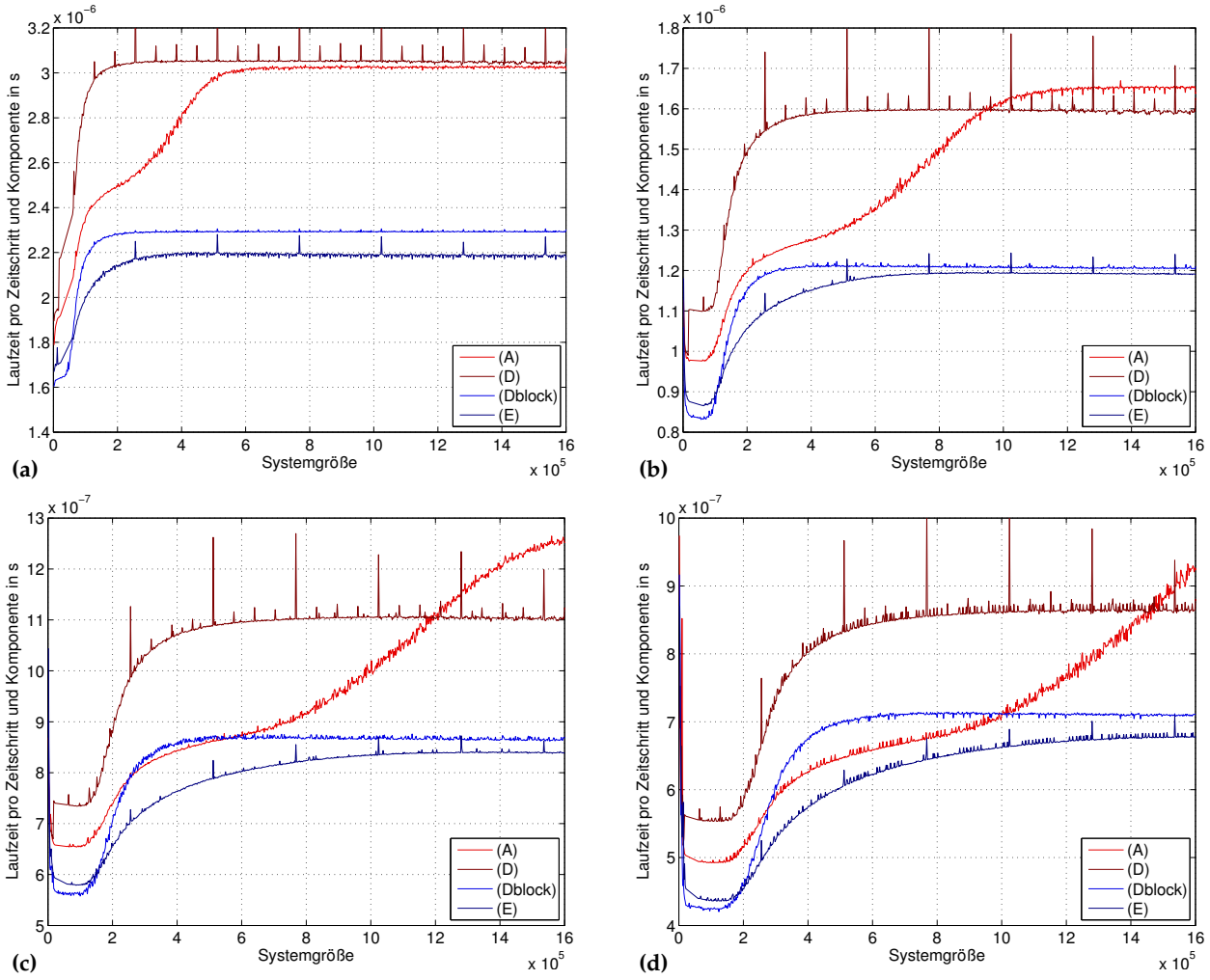


Abb. 3.13: Laufzeit der allgemeinen parallelen Implementierungen für gemeinsamen Adreßraum pro Zeitschritt und Komponente in Abhängigkeit von der Systemgröße auf einem Itanium-2-System für das Testproblem MEDAKZO und das Verfahren DOPRI8(7). **(a)** 1 Thread, **(b)** 2 Threads, **(c)** 3 Threads und **(d)** 4 Threads.

Implementierung (D) aufgrund des veränderten Verhaltens von `malloc()` ist jedoch stärker als im Fall der sequentiellen Implementierungen. Dies läßt vermuten, daß das Ausrichten der Speicherblöcke durch `malloc()` nicht nur Konfliktfehlzugriffe zwischen Datenzugriffen provoziert, sondern außerdem auch Konflikte mit Instruktionszugriffen verstärkt.

Für alle untersuchten Testprobleme führt die parallele Ausführung der Implementierungen zu einer Laufzeitverbesserung gegenüber den sequentiellen Implementierungen. Die erreichten Speedups sind dabei abhängig von der Systemgröße. Bezogen auf die Laufzeit der jeweils entsprechenden sequentiellen Implementierung erreicht jede Implementierung in bestimmten Bereichen der Systemgröße superlineare Speedups. Bezieht man sich auf die Laufzeit der *schnellsten* sequentiellen Implementierungsvariante können nur einige Implementierungen superlineare Speedups erreichen, und dies auch nur für eine kleinere Menge von Systemgrößen. Eine genaue quantitative Auswertung der Speedup-Werte wäre aufgrund der Interferenzen zwischen Daten- und Instruktionszugriffen nicht allgemeingültig, da bereits geringfügige Veränderungen des Programmcodes oder Verschiebungen des Programmcodes an eine andere Speicheradresse zu anderen Aussagen führen würden.

BRUSS2D-MIX und MEDAKZO besitzen nach Definition 2.2 eine relativ kleine Zugriffsdistanz von $d(\mathbf{f}) = 2N = 2\sqrt{n/2}$ bzw. $d(\mathbf{f}) = 2$. Die Zugriffsdistanz von BRUSS2D-ROW ist dagegen mit $d(\mathbf{f}) = N^2 = n/2$ relativ groß. Bei genauerer Betrachtung wird jedoch deutlich, daß die Funktionsauswertung für

BRUSS2D-ROW nur ein einziges Element des Argumentvektors in einer Distanz von N^2 benutzt; alle anderen zugegriffenen Vektorelemente liegen innerhalb einer Distanz von N. Das Verhalten von BRUSS2D-ROW zeigt deshalb einige Eigenschaften, wie man sie von Problemen mit einer Zugriffsdistanz von N erwarten würde. Aus diesem Grund können für alle Testprobleme Verkleinerungen der Arbeitsmengen, die im sequentiellen Fall aus der Wiederverwendung vollständiger Vektoren resultierten, um einen Faktor von rund $1/p$ beobachtet werden. Beispielsweise ist für die parallele Implementierung (D) bei Verwendung des Testproblems BRUSS2D-MIX bei der Ausführung mit nur einem Thread bei einer Systemgröße von ca. 80 000 ein Ansteigen der normierten Laufzeit erkennbar, das auf ein Herausfallen einer Arbeitsmenge von $10n$ aus dem L3-Cache hindeutet. Ab einer Systemgröße von ca. 260 000, was einer Arbeitsmenge von $3n$ entspricht, findet kein weiteres Ansteigen der normierten Laufzeit statt. Erhöht man die Anzahl der Threads auf 2 bzw. 4, verdoppeln bzw. vervierfachen sich die Systemgrößen, an denen diese Ereignisse eintreten, in etwa auf 160 000 und 500 000 bzw. 300 000 und 1 100 000, was die annähernde Halbierung bzw. Viertelung der entsprechenden Arbeitsmengen belegt.

Von der Verkleinerung der Arbeitsmengen profitieren besonders stark solche Implementierungen, die sehr große Arbeitsmengen besitzen. Beispielsweise erreicht die normierte Laufzeit der Implementierung (A) für BRUSS2D-MIX bei Ausführung mit nur einem Thread ihr Maximum erst ab einer Systemgröße von rund 600 000. Durch Erhöhung der Anzahl der Threads auf 4 wird dieser Maximalwert erst oberhalb einer Systemgröße von 2 000 000 erreicht, wodurch sich für den großen Bereich zwischen diesen beiden Systemgrößen durch die Parallelisierung eine erhebliche Laufzeitverbesserung gegenüber der Ausführung mit einem Thread ergibt, die über den Faktor 4 hinausgeht. Zwar verkleinern sich auch die Arbeitsmengen der übrigen Implementierungen im gleichen Verhältnis, dennoch erreicht Implementierung (A) für MEDAKZO bei Ausführung mit 4 Threads für Systemgrößen zwischen ca. 300 000 und 1 000 000 eine geringere Laufzeit als (Dblock), obwohl Implementierung (A) bei Ausführung mit nur einem Thread immer deutlich langsamer als (Dblock) ist.

Skalierbarkeit für bis zu 32 Prozessoren

Um die Skalierbarkeit für eine größere Anzahl von Prozessoren zu untersuchen, wurden Laufzeitexperimente auf einem Knoten des Supercomputers JUMP durchgeführt. Hierbei handelt es sich um einen Server des Typs IBM p690 mit 32 Power4+-Kernen und einer dreistufigen Cache-Hierarchie. Für eine detaillierte Beschreibung siehe Anhang A.

Abbildung 3.14 zeigt die auf diesem System für BRUSS2D-MIX mit dem Verfahren DOPRI5(4) für die Problemgrößen $N = 250$, $N = 500$ und $N = 1000$ gemessenen Speedup-Werte. Als Blockgröße für die Implementierung (Dblock) wurde dabei $B = 2N$ verwendet. Die erzielten Speedup-Werte waren bei diesen Experimenten um so größer, je höher die Problemgröße gewählt wurde. Zwar nimmt mit zunehmender Problemgröße auch der Einfluß der Barrier-Operationen auf die Gesamtlaufzeit ab, die z. T. erheblichen Laufzeitunterschiede zwischen den Implementierungsvarianten deuten jedoch darauf hin, daß das Lokalitätsverhalten den wichtigsten Einfluß auf die Skalierbarkeit ausübt.

Für $N = 250$ zeigen alle Implementierungen für bis zu 16 Prozessoren ein ähnlich gutes Skalierbarkeitsverhalten, wobei Speedups zwischen 9,2 und 9,6 erreicht werden, was einer Effizienz zwischen 0,57 und 0,60 entspricht. Bei Erhöhung der Prozessorzahl auf 24 können nur die Implementierungen (D) und (Dblock) ihren Speedup steigern. Diese beiden Implementierungen erreichen in diesem Fall einen Speedup von 10,8 bzw. 10,9, was einer Effizienz von rund 0,45 entpricht. (A) und (E) liefern auf 24 Prozessoren nur einen Speedup von 9,0 bzw. 9,5 und verschlechtern somit ihre Effizienz. Bei einer weiteren Erhöhung der Prozessorzahl auf 32 verschlechtern sich die Speedups aller Implementierungen.

Die Erhöhung der Problemgröße auf $N = 500$ führt zu einer deutlichen Verbesserung der Skalierbarkeit. Mit Ausnahme der Implementierung (E) können alle Implementierungen ihren Speedup mit steigender Prozessorzahl bis zur maximal verfügbaren Zahl von 32 Prozessoren verbessern. Jedoch erhöhen sich die Speedups beim Übergang von 24 auf 32 Prozessoren nur geringfügig. Für Implementierung (E) wurde der maximale Speedup bei Verwendung von 24 Prozessoren gemessen. Insgesamt bieten für diese Eingabegröße alle Implementierung eine ähnlich gute Skalierbarkeit. Die maximalen Speedups aller Implementierungen liegen zwischen 17,3 und 18,0, was für 32 Prozessoren einer Effizienz zwischen 0,54 und 0,56 entspricht. Werden weniger Prozessoren eingesetzt, arbeiten die Implementierungen jedoch wesentlich effizienter. Für 2 und für 4 Prozessoren erreicht Implementierung (E) sogar ideale Speedup-Werte von 2,0 bzw. 4,0.

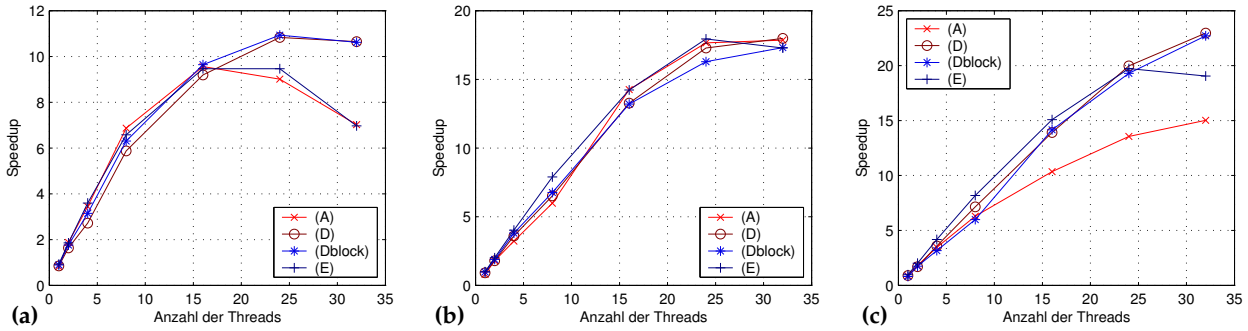


Abb. 3.14: Laufzeit der allgemeinen parallelen Implementierungen für die Lösung des Testproblems BRUSS2D-MIX mit $H = 4,0$ und dem Verfahren DOPRI5(4) auf einem Knoten des Supercomputers JUMP für verschiedene Problemgrößen. **(a)** $N = 250$, **(b)** $N = 500$ und **(c)** $N = 1000$.

Die höchsten Speedups wurden für die Problemgröße $N = 1000$ gemessen. Die Implementierungen (D) und (Dblock) können in diesem Fall ihre Speedups bei Erhöhung der Prozessorzahl bis auf einen Wert von 23,0 bzw. 22,7 bei Verwendung von 32 Prozessoren steigern. Dies entspricht einer Effizienz von 0,72 bzw. 0,71. Der maximale Speedup für Implementierung (E) mit einem Wert von 19,7 wurde dagegen bei Verwendung von nur 24 Prozessoren gemessen. Für diese Problemgröße erreicht Implementierung (E) für bis zu 8 Prozessoren sogar superlineare Speedups. So konnte z. B. für 4 Prozessoren ein Speedup von 4,2 und für 8 Prozessoren ein Speedup von 8,2 gemessen werden. Implementierung (A) kann zwar ihren Speedup für bis zu 32 Prozessoren stetig steigern, arbeitet aber ineffizienter als die übrigen Implementierungen und erreicht so nur einen maximalen Speedup von 15,0.

3.4.4 Zusammenfassung

Moderne Parallelrechner mit gemeinsamem Adreßraum besitzen häufig eine NUMA-Architektur mit tiefer Speicherhierarchie. Die Kommunikation zwischen Kontrollflüssen erfolgt im wesentlichen über den gemeinsamen Adreßraum mit Hilfe von Speicherzugriffsoperationen. Im Unterschied zu Parallelrechnern mit verteiltem Adreßraum übt deshalb auf diesen Systemen das Lokalitätsverhalten einen entscheidenden Einfluß auf die Skalierbarkeit eines parallelen Programms aus. Programme für Parallelrechner mit gemeinsamem Adreßraum verwenden häufig mehrere Threads, die über gemeinsame Variablen bzw. gemeinsam genutzte Datenstrukturen miteinander kommunizieren. Standardschnittstellen für diese Art der Programmierung sind OpenMP (OpenMP API 2.5 2005) und POSIX Threads (Butenhof 1997).

Parallele Implementierungen eingebetteter Runge-Kutta-Verfahren können in Anlehnung an das Rahmenprogramm (Abb. 3.1) unter Verwendung solcher Programmierschnittstellen relativ einfach realisiert werden. Dazu werden die gleichen Datenstrukturen verwendet, wie für die sequentielle Implementierung, wodurch die parallelen Implementierungen den gleichen Speicherplatzbedarf wie diese besitzen. Zur Parallelisierung führen die beteiligten Threads jedoch jeweils nur einen bestimmten Teil der Schleifeniterationen über die Systemdimension aus. Um eine gute räumliche und zeitliche Lokalität zu erzielen, ist dabei eine blockweise Verteilung der Schleifeniterationen vorteilhaft. Zur Vermeidung von Sharing-Effekten kann die Verteilung auf Cachezeilenbasis erfolgen. Besteht die Möglichkeit, Threads an Prozessoren zu binden, kann dadurch ebenfalls eine höhere Lokalität erzielt werden. Die Datenabhängigkeiten der Funktionsauswertungen erfordern eine diesbezügliche Synchronisation der Threads. Dies kann durch die Ausführung von ein bzw. zwei Barrier-Operationen in jeder Stufe realisiert werden.

Ausgehend von den vier sequentiellen Implementierungsvarianten (A), (E), (D) und (Dblock), die in Kapitel 2 vorgestellt wurden, wurden in diesem Kapitel vier korrespondierende parallele Implementierungen für gemeinsamen Adreßraum entwickelt, die wesentliche Eigenschaften des Lokalitätsverhaltens mit der jeweiligen Ausgangsimplementierung gemeinsam haben. Durch die Parallelisierung der Schleifen über die Systemdimension wird von jedem Thread jedoch nur noch auf einen Teil der Komponenten der verwendeten Vektoren zugegriffen. Da jeder der p Threads maximal $\lceil n/p \rceil$ Iterationen ausführt, wird in vielen Fällen auch nur auf $\lceil n/p \rceil$ Komponenten der betreffenden Vektoren zugegriffen. Die Arbeitsräume dieser Schleifen verkleinern sich somit um einen Faktor von $1/p$. Die Parallelisierung führt also zu einer Verbesserung

des Lokalitätsverhaltens mit zunehmender Anzahl von Threads. Für bestimmte Problemgrößen, die dazu führen, daß erst durch die Aufteilung der zu verarbeitenden Daten wichtige Arbeitsmengen im Cache gespeichert werden können, sind deshalb superlineare Speedups möglich. Finden in einer parallelen Schleife über die Systemdimension jedoch Funktionsauswertungen statt, so werden – abhängig von der Zugriffsdistanz – mehr als $\lceil n/p \rceil$ Komponenten des jeweiligen Argumentvektors benutzt. Davon betroffen sind vor allem die Implementierungen (D) und (Dblock), in denen alle Argumentvektoren $\mathbf{w}_1, \ldots, \mathbf{w}_s$ getrennt gespeichert werden.

Laufzeitexperimente mit dem Testproblem BRUSS2D-MIX auf einem mit 32 Prozessoren ausgestatteten Knoten des Supercomputers JUMP zeigen, daß auf realen Rechnersysteme für kleine Prozessorzahlen eine sehr effiziente Ausführung möglich ist. In einem Experiment konnten für Implementierung (E) für bis zu 8 Prozessoren sogar superlineare Speedups beobachtet werden. Allgemein verbesserte sich auf dieser Maschine in den durchgeführten Experimenten die Skalierbarkeit mit zunehmender Problemgröße. Für die höchste betrachtete Problemgröße, $N = 1000$, konnten Speedups bis zu einem Wert von 23,0 gemessen werden. Da z. T. deutliche Unterschiede im Skalierbarkeitsverhalten der verschiedenen Implementierungsvarianten auftraten, obwohl sich diese hauptsächlich durch ihr Lokalitätsverhalten unterscheiden, bestätigen die Laufzeitexperimente die Wichtigkeit der Lokalitätsoptimierung für die möglichst optimale Nutzung moderner Parallelrechner mit gemeinsamem Adreßraum.

3.5 Hybride Implementierung für SMP-Cluster

Viele moderne parallele Rechnersysteme sind sogenannte SMP-Cluster. Darunter versteht man den Zusammenschluß mehrerer SMP-Systeme zu einem Clusterverbund. Der Vorteil eines solchen Aufbaus liegt darin, daß Hardwarekomponenten genutzt werden können, die auch außerhalb des Höchstleistungsrechnens eingesetzt werden und deshalb preiswert – im Vergleich zu speziell entworfenen Supercomputerlösungen – angeschafft werden können. Steht ein ausreichend schnelles Verbindungsnetzwerk zwischen den Knoten zur Verfügung, können SMP-Cluster den auf ihnen ausgeführten Programmen eine sehr gute Skalierbarkeit ermöglichen, da diese innerhalb der SMP-Knoten über den schnellen gemeinsamen Speicher kommunizieren können und nur ein Teil der Kommunikation über das langsamere Verbindungsnetzwerk abgewickelt werden muß.

Davon können auch Programme profitieren, die ursprünglich für einen verteilten Adreßraum entwickelt wurden. Denn selbst wenn diese auch innerhalb der SMP-Knoten über Netzwerksockets kommunizieren, kann diese Kommunikation innerhalb eines Knotens schneller abgewickelt werden, als wenn die auszutauschenden Nachrichten tatsächlich ein Netzwerk durchlaufen würden. Verwenden die Programme eine Standardschnittstelle wie beispielsweise MPI (MPI Forum 1994, 1995, 1997), kann jedoch durch den Einsatz speziell für SMP-Cluster optimierter Implementierungen dieser Schnittstellen eine deutlich bessere Performance erreicht werden. Allerdings nutzen auch die optimierten Implementierungen Prozesse zur Realisierung der Kontrollflüsse des Programms, und jeder dieser Prozesse erhält vom Betriebssystem einen separaten virtuellen Adreßraum. Die Kommunikation zwischen Prozessen kann also auch innerhalb eines Knotens nicht direkt durch Speicherzugriffsoperationen realisiert werden, sondern muß umständlich über spezielle, vom Betriebssystem bereitgestellte Mechanismen erfolgen. Diesen Nachteil beseitigen speziell für den Einsatz auf SMP-Clustersystemen entworfene Kommunikationsbibliotheken wie z. B. vShark (Hunold u. Rauber 2005), indem sie Threads zur Realisierung der Kontrollflüsse nutzen und somit durch Ausnutzung des gemeinsamen virtuellen Adreßraumes den Nachrichtenaustausch sehr effizient durchführen können.

Für eine bestmögliche Ausnutzung der speziellen Architektur von SMP-Clustersystemen sollten jedoch bereits beim Entwurf einer Anwendung die spezifischen Eigenschaften dieser Systeme berücksichtigt werden. Dazu bietet sich die Nutzung eines hybriden Programmiermodells an, das eine Standardschnittstelle wie z. B. MPI zur Kommunikation zwischen verschiedenen Knoten verwendet und innerhalb eines Knotens durch die Verwendung von Threads den direkten Zugriff auf gemeinsam genutzte Datenstrukturen ermöglicht. Ein expliziter Nachrichtenaustausch zwischen Kontrollflüssen, die auf dem selben Knoten ausgeführt werden, ist dann nicht mehr notwendig.

In Laufzeitexperimenten mit parallelen Implementierungen eingebetteter Runge-Kutta-Verfahren für verteilten Adreßraum, die ausschließlich MPI zur Kommunikation nutzen, konnten diese Implementie-

rungen in der Regel keine gute Skalierbarkeit erreichen (vgl. Abschnitt 3.3.4). So konnten auf einem SMP-Knoten des Supercomputers JUMP nicht mehr als 16 Prozessoren sinnvoll genutzt und nicht einmal ein Speedup von 4 erreicht werden. Eine Ausdehnung der Messung auf weitere Knoten kam daher nicht in Frage. Implementierungsvarianten für gemeinsamen Adreßraum auf Basis von POSIX Threads konnten auf diesem System dagegen bis zu 32 Prozessoren effizient nutzen und einen Speedup bis zu 23,0 erreichen (siehe Abschnitt 3.4.3). Eine hybride Implementierung eingebetteter Runge-Kutta-Verfahren, die beide Programmiermodelle kombiniert erscheint daher vielversprechend.

Dieser Ansatz wurde in der von Behrend (2005) der Universität Bayreuth vorgelegten Bachelor-Thesis verfolgt. Im Rahmen dieser Arbeit wurden u. a. Varianten der Implementierungen (A) und (D) erstellt und untersucht. Diese nutzen MPI zur Kommunikation zwischen SMP-Knoten, indem sie die Erzeugung einer Anzahl von MPI-Prozessen gestatten, die auf Basis einer blockweisen Verteilung die Schleifen über die Systemdimension parallel ausführen. Dazu nutzen sie in jeder Stufe eine Allgather-Operation, um die von den einzelnen Prozessen berechneten Teile des jeweiligen Argumentvektors zusammenzufügen und allen Prozessen zur Verfügung zu stellen. Zur Kommunikation innerhalb der Knoten nutzen die Implementierungen POSIX Threads. Dazu erzeugt jeder MPI-Prozeß eine Anzahl von Threads, die parallel die Iterationen der Schleifen über die Systemdimension ausführen, die dem übergeordneten MPI-Prozeß zugeordnet wurden. Die Verteilung der Iterationen auf die teilnehmenden Threads erfolgt dabei wiederum blockweise. Zur Synchronisation der Threads werden wie bei den Implementierungen für gemeinsamen Adreßraum Barrier-Operationen genutzt. Durch diese muß unter anderem sichergestellt werden, daß die Allgather-Operation erst dann ausgeführt wird, wenn alle Threads die Berechnung der ihnen zugeordneten Komponenten des jeweiligen Argumentvektors abgeschlossen haben, und daß erst dann mit der Ausführung von Funktionsauswertungen begonnen wird, wenn die Allgather-Operation abgeschlossen ist. Um hierzu nicht den Overhead von zwei Barrier-Operationen in Kauf nehmen zu müssen, kann die Allgather-Operation mit einer Barrier-Operation verschmolzen werden, so daß der letzte an der Barrier eintreffende Thread die Allgather-Operation ausführt und erst danach veranlaßt, daß alle Threads die Ausführung fortsetzen. Der Gewinn dieses Verschmelzens der beiden Operationen ist jedoch oft nur gering, da der Overhead der Barrier-Operation häufig gering gegenüber der Laufzeit der Allgather-Operation ist. Laufzeitexperimente auf dem Supercomputer JUMP sowie auf einem vom Lehrstuhl für Angewandte Informatik 2 der Universität Bayreuth betriebenen Clustersystem, das aus 32, über Infiniband, Gigabit Ethernet und Fast Ethernet vernetzten, Zwei-Wege-Opteron-SMP-Knoten besteht, zeigen, daß diese hybriden Implementierungsvarianten zwar nicht in allen, jedoch in vielen Fällen zu einer besseren Laufzeit führen als die zum Vergleich herangezogenen Implementierungen für verteilten Adreßraum.

3.6 Dynamische Lastbalancierung

Bei allen bisher betrachteten parallelen Implementierungen findet in jeder Stufe eine Synchronisation der beteiligten Kontrollflüsse statt: Im Fall eines verteilten Adreßraumes müssen vor jeder Funktionsauswertung die parallel berechneten Teile des benötigten Argumentvektors mittels einer Multibroadcastoperation allen beteiligten Kontrollflüssen bekannt gemacht werden. Besitzen die beteiligten Kontrollflüsse einen gemeinsamen Adreßraum, so müssen an entsprechender Stelle Barrier-Operationen ausgeführt werden, um die Berechnung und den Zugriff auf den jeweiligen Argumentvektor zu schützen. Infolgedessen hängt die Effizienz der parallelen Implementierungen wesentlich davon ab, daß die beteiligten Kontrollflüsse diese Synchronisationspunkte möglichst *gleichzeitig* erreichen, da andernfalls Wartezeiten entstehen. In der Praxis können jedoch verschiedene Umstände dazu führen, daß die erforderliche Synchronität der Kontrollflüsse nicht gegeben ist:

1. Das parallele Rechnersystem kann aus Prozessoren mit unterschiedlicher Geschwindigkeit aufgebaut, d. h. heterogen sein.

2. Die Funktionsauswertungszeiten der einzelnen Komponenten des Differentialgleichungssystems können unterschiedlich groß sein, da unterschiedlich viele Instruktionen für die Funktionsauswertung benötigt werden.

3. Die durch Speicherzugriffe bedingten Wartezeiten können stark variieren, abhängig davon, bis zu welcher Ebene die Speicherhierarchie durchlaufen werden muß. Besonders groß sind die Unterschiede

auf NUMA-Systemen (NUMA: englisch für *non-uniform memory access*), auf denen ein Speicherzugriff – im günstigsten Fall eines L1-Cache-Treffers – z. B. innerhalb eines Taktes abgeschlossen sein kann, im schlechtesten Fall jedoch eine komplexe Hierarchie von Switches durchlaufen werden muß, um die Daten aus einem Speichermodul eines anderen Prozessors zu transferieren, wozu mehrere 1000 Takte erforderlich sein können.

4. Das Programm kann vom Betriebssystemscheduler unterbrochen werden, um andere Prozesse oder Systemdienste auszuführen.

5. Auf SMT-Systemen konkurrieren die Threads um die Funktionseinheiten des Prozessors. Sind nicht genügend Funktionseinheiten vorhanden, um alle Threads gleichzeitig ausführen zu können, müssen einige Threads zurückgestellt werden.

Es erscheint deshalb sinnvoll, Techniken zu untersuchen, die eine Verringerung der Wartezeiten bezüglich der Barrier-Operationen anstreben. Hierzu bieten sich dynamische Lastbalancierungstechniken an, welche die auszuführenden Berechnungen zur Laufzeit auf die beteiligten Kontrollflüsse verteilen.

3.6.1 Analyse der Funktionsauswertungszeiten verschiedener Testprobleme

Bevor wir uns mit der Realisierung geeigneter Lastbalancierungstechniken beschäftigen, betrachten wir zunächst anhand der Beispiele EMEP, BRUSS2D, MEDAKZO und STARS, wie stark sich die Kosten der Funktionsauswertung für die einzelnen Komponenten für diese Beispiele unterscheiden. Abbildung 3.15 zeigt für vier Testprobleme, wieviele Instruktionen und wieviele Taktzyklen für die Auswertung der einzelnen Komponenten erforderlich sind. Eine genaue Beschreibung dieser Testprobleme wird in Anhang B gegeben. Die Messung der Daten erfolgte mit Hilfe der PCL-Bibliothek (Berrendorf u. Mohr 2003) auf einer Workstation vom Typ Fujitsu-Siemens SCENIC W600, die mit einem Pentium-4-Prozessor mit einer Taktfrequenz von 3 GHz ausgestattet ist (siehe Anhang A).

EMEP ist ein Problem der Dimension 66, welches mehrere chemische Reaktionen beschreibt. Die Formeln zur Berechnung der einzelnen Komponenten der rechten Seite besitzen eine sehr unterschiedliche Komplexität, wodurch ein hoher Grad von Irregularität entsteht. Die Anzahl der Instruktionen pro Funktionsauswertung liegt zwischen etwa 77 und 4085, im Mittel liegt sie bei ca. 847. Die hierfür erforderliche Laufzeit liegt etwa zwischen 202 Takten (68 ns) und 7860 Takten (2620 ns). Der Mittelwert beträgt 1650 Takte (550 ns). Die irreguläre Verteilung der Funktionsauswertungszeiten macht dieses Problem für die Untersuchung von Lastbalancierungsverfahren sehr interessant.

BRUSS2D wurde durch Diskretisierung mittels Linienmethode über einem $N \times N$-Gitter aus einem zweidimensionalen System von PDEs mit zwei abhängigen Variablen U und V abgeleitet. Daraus resultiert eine Berechnungsvorschrift, die durch die Zugriffsstruktur des zur Diskretisierung verwendeten 5-Punkt-Sterns gekennzeichnet ist. Unterschiedliche Funktionsauswertungskosten für die einzelnen Komponenten resultieren aus folgenden Eigenschaften:

- unterschiedliche Berechnungsvorschriften für U und V,
- unterschiedliche Berechnungsvorschriften für innere Gitterpunkte und Punkte in den verschiedenen Randbereichen,
- unterschiedliche Anzahl von Vergleichen für die Fallunterscheidung, ob ein Punkt zu U oder zu V gehört, ob er ein innerer Punkt ist oder ob er zu einem Randbereich gehört.

Es wurden zwei Implementierungen des Testproblems BRUSS2D realisiert, die jeweils unterschiedliche Linearisierungen der Komponenten verwenden:

$$U_{11}, U_{12}, \ldots U_{NN}, V_{11}, V_{12}, \ldots, V_{NN} \qquad \text{(zeilenorientiert, BRUSS2D-ROW)},$$
$$U_{11}, V_{11}, U_{12}, V_{12}, \ldots, U_{ij}, V_{ij}, \ldots U_{NN}, V_{NN} \qquad \text{(gemischt-zeilenorientiert, BRUSS2D-MIX)}.$$

Die beiden Implementierungen unterscheiden sich auch in der Fallunterscheidungsstrategie zur Erkennung der Randkomponenten. Im Fall von BRUSS2D-ROW wurde versucht, innere Komponenten mit möglichst wenigen Vergleichen zu identifizieren. Dadurch erhöht sich allerdings die Anzahl der zur Identifizierung der Randkomponenten notwendigen Vergleiche. Dies ist in Abb. 3.15 (b) sehr gut zu erkennen. Für

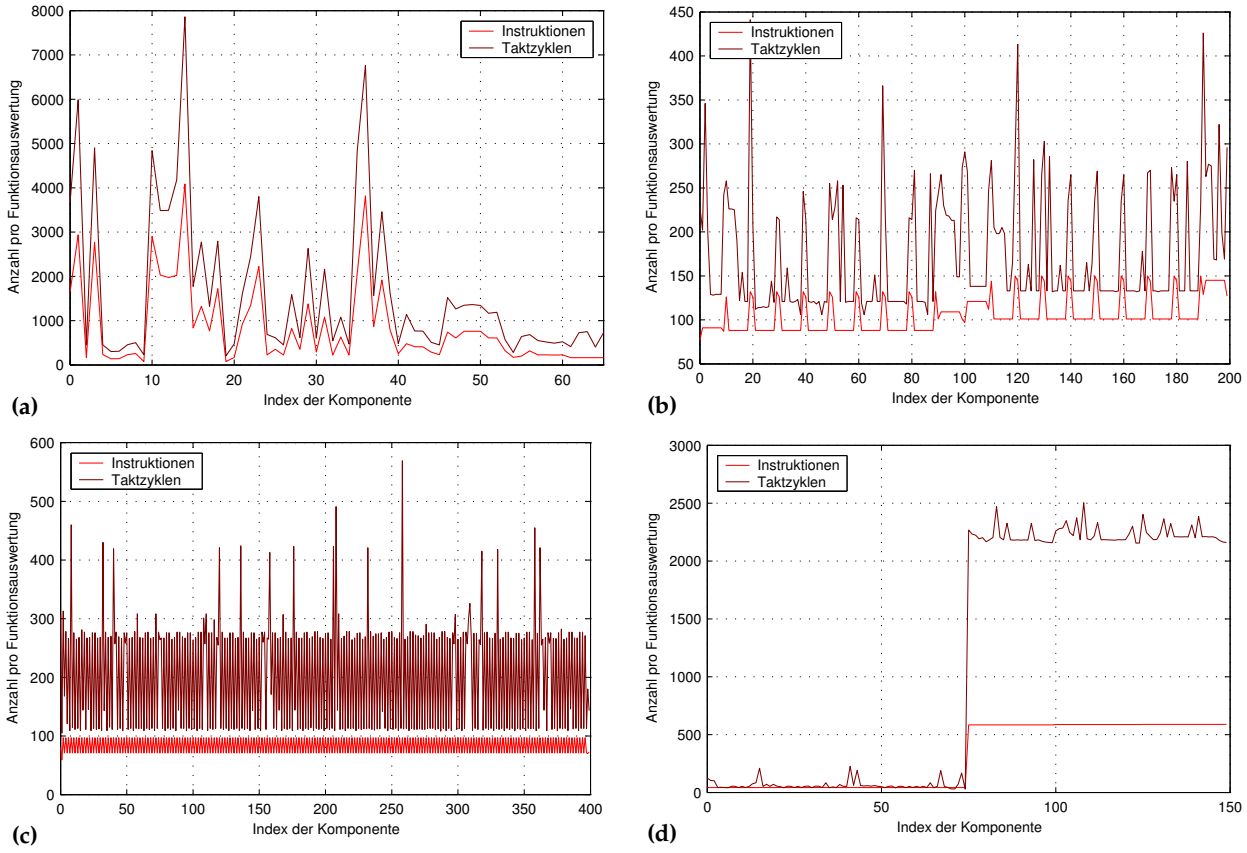


Abb. 3.15: Anzahl der Instruktionen und Anzahl der Taktzyklen, die für die Auswertung der einzelnen Komponenten des Differentialgleichungssystems erforderlich sind. **(a)** EMEP, **(b)** BRUSS2D-ROW, $N = 10$, **(c)** MEDAKZO, $N = 200$, **(d)** STARS-CON, $N = 25$.

BRUSS2D-MIX wurde dagegen ein nahezu ausgeglichener Entscheidungsbaum angestrebt. Bezüglich der Anzahl der ausgeführten Instruktionen scheint die für BRUSS2D-MIX verwendete Fallunterscheidungsstrategie geringfügige Vorteile zu besitzen. Während die Funktionsauswertungen für BRUSS2D-MIX für $N = 10$ etwa zwischen 84 und 91, im Mittel ca. 90 Instruktionen verwenden, liegt die benötigte Anzahl von Instruktionen für BRUSS2D-ROW zwischen 77 und 150, im Mittel bei ca. 105. Betrachtet man dagegen die Anzahl der benötigten Takte, schneidet BRUSS2D-ROW besser ab. Im Experiment benötigte BRUSS2D-ROW etwa zwischen 105 und 442 Takten (35 und 148 ns), im Mittel ca. 176 Takte (59 ns). BRUSS2D-MIX dagegen benötigte zwischen 123 und 416 Takten (41 und 139 ns). Im Mittel waren es 178 Takte (60 ns). Bei einer Erhöhung der Systemgröße durch Verfeinerung des Diskretisierungsgitters ändert sich die Anzahl der im Mittel ausgeführten Instruktionen pro Funktionsauswertung einer Komponente für BRUSS2D-MIX nicht. Für BRUSS2D-ROW ergibt sich dagegen eine geringfügige Verbesserung, da sich der Anteil der Randkomponenten an der Gesamtzahl der Komponenten verringert. Für beide Linearisierungen ist ein geringer Anstieg der Ausführungszeiten beobachtbar, der Folge des Anstiegs der Speicherzugriffszeiten aufgrund einer gestiegenen Anzahl von Cache-Fehlzugriffen ist.

Das Testproblem MEDAKZO resultiert aus der Diskretisierung eines Systems eindimensionaler PDEs mit zwei abhängigen Variablen mittels Linienmethode. Die Anordnung der ODE-Komponenten erfolgt durch eine Verschachtelung der den beiden abhängigen Variablen zugehörigen Komponenten. Infolge der unterschiedlichen Berechnungsvorschriften für die beiden abhängigen Variablen unterscheiden sich die Funktionsauswertungszeiten für gerade und ungerade Komponenten. Die Anzahl der pro Funktionsauswertung benötigten Instruktionen liegt zwischen 59 und 97, im Mittel bei ca. 84. Die Laufzeiten schwanken zwischen 105 Takten (35 ns) und 570 Takten (190 ns). Im Mittel liegen sie bei 205 Takten (69 ns).

STARS beschreibt die Bewegung von Sternen unter Einwirkung der Gravitation. Die Irregularität dieses Testproblems resultiert daraus, daß die Bewegungsgleichung die Ordnung zwei besitzt und somit eine

Ordnungsreduzierung durch Substitution und Verdopplung der Systemdimension notwendig ist. Da die Bewegungsgleichungen die Wechselwirkung eines Sterns mit allen anderen Sternen berücksichtigen müssen, wächst allerdings der Aufwand zur Berechnung der Komponenten der zweiten Ableitung linear mit der Anzahl der Sterne. Die Funktionsauswertungen für STARS benötigen für $N = 25$ nur 42 Instruktionen zur Berechnung der ersten Ableitung, die in der Regel jeweils innerhalb eines Taktes ausgeführt werden können, und 583 bis 588 Instruktionen zur Berechnung der Beschleunigungen, wofür maximal ca. 2500 Taktzyklen erforderlich sind. Es wurden zwei verschiedene Linearisierungen implementiert. STARS-CON speichert die beiden Ableitungen zusammenhängend in aufeinanderfolgenden Speicherblöcken ab, d. h., die Geschwindigkeitskomponenten (v_{ix}, v_{iy}, v_{iz}) und die Beschleunigungskomponenten (a_{ix}, a_{iy}, a_{iz}) werden wie folgt angeordnet:

$$v_{1x}, \ldots, v_{Nx}, v_{1y}, \ldots, v_{Ny}, v_{1z}, \ldots, v_{Nz}, a_{1x}, \ldots, a_{Nx}, a_{1y}, \ldots, a_{Ny}, a_{1z}, \ldots, a_{Nz} \ .$$

Bei Verwendung einer blockweisen Datenverteilung führt diese Anordnung der Komponenten zu einem sehr großen Lastungleichgewicht. Eine bezüglich der Lastverteilung günstigere Anordnung der Komponenten realisiert STARS-MIX:

$$v_{1x}, a_{1x}, \ldots, v_{Nx}, a_{Nx}, v_{1y}, a_{1y}, \ldots, v_{Ny}, a_{Ny}, v_{1z}, a_{1z}, \ldots, v_{Nz}, a_{Nz} \ .$$

3.6.2 Allgemeine Ansätze zur Lastbalancierung irregulärer Algorithmen

Als *irreguläre Algorithmen* bezeichnet man solche Algorithmen, deren Verhalten von der Eingabe abhängig ist. Eine statische Verteilung der Berechnungen während des Programmentwurfs oder während der Übersetzung kann deshalb nicht für alle möglichen Eingaben eine optimale Laufzeit liefern. Stattdessen ist es erforderlich, die Berechnungen während der Laufzeit – wenn also die Eingabe bekannt ist – an die beteiligten Kontrollflüsse zuzuweisen. Eingebettete Runge-Kutta-Verfahren sind in diesem Sinne irreguläre Algorithmen, da die Funktionsauswertungszeiten der einzelnen Komponenten des Differentialgleichungssystems unterschiedlich sein können und wir davon ausgehen, daß das zu lösende Anfangswertproblem erst beim Programmstart ausgewählt, d. h. als Eingabe spezifiziert wird.

Das Problem der Lastbalancierung irregulärer Algorithmen ist bereits seit einigen Jahren Gegenstand intensiver Forschung. Können die von einem Programm durchzuführenden Berechnungen in Form von parallelen Schleifen mit unabhängigen Iterationen formuliert werden, ist eine Verteilung der Schleifeniterationen auf die vorhandenen Prozessoren mit Hilfe von Schleifen-Scheduling-Verfahren möglich, siehe z. B. Tabirca u. a. (2004), Polychronopoulos u. Kuck (1987), Carino u. Banicescu (2005). Aufgrund der großen Bedeutung dynamischer Lastbalancierungsverfahren für parallele Schleifen wurden solche Verfahren auch in OpenMP (OpenMP API 2.5 2005) und High Performance Fortran (HPF, High Performance Fortran Forum 1997) integriert. Eine alternative anwendungsunabhängige Möglichkeit der Lastbalancierung stellt die Zerlegung des Ursprungsproblems in möglichst unabhängige Unterprobleme (*Tasks*) dar, welche statisch oder dynamisch den verfügbaren Prozessoren zur Ausführung zugeordnet werden. Statische Ansätze zum Scheduling von taskbasierten Anwendungen verwenden oft Taskgraphen zur Beschreibung der Abhängigkeiten zwischen den Tasks. Dieser statische Ansatz wird u. a. von Beaumont u. a. (2002), El-Rewini u. a. (1995), Norman u. Thanisch (1993), Turek u. a. (1992), Bampis u. Kononov (2001) und Lepère u. a. (2002) verfolgt.

Ein dynamisches Scheduling von Einzelprozessortasks kann z. B. mit Hilfe einer spezialisierten Datenstruktur (*Taskpool*) gesteuert werden, in die jeder Prozessor Tasks einfügen und aus der jeder Prozessor Tasks entnehmen kann. Die Lastbalancierung wird erreicht, indem die Prozessoren bei Bedarf vom Taskpool Arbeit in Form der Tasks anfragen oder zugeteilt bekommen, so daß alle Prozessoren zu jeder Zeit mit der Ausführung von Tasks beschäftigt sind. Die für die Realisierung eines Taskpools verwendeten Basisdatenstrukturen und Zugriffsverfahren haben einen großen Einfluß auf die resultierende parallele Ausführungszeit der betrachteten Anwendung. Typische Basisdatenstrukturen sind verschiedene Formen von *Warteschlangen*, die z. B. als *verteilte Warteschlangen* vorrangig von einem bestimmten Prozessor zugegriffen werden oder als *zentrale Warteschlangen* keinem bestimmtem Prozessor zugeordnet sind und von allen Prozessoren mit etwa gleicher Häufigkeit zugegriffen werden. Eine weitere Möglichkeit ist eine hierarchische Anordnung der Datenstrukturen (siehe z. B. Dandamudi u. Ayachi 1999). Der Zugriff auf die Datenstrukturen und die Lastbalancierung kann auf verschiedene Weise realisiert werden (Kumar u. a.

1994). Empfängerbasierte Verfahren führen die Lastbalancierung dann aus, wenn ein Prozessor keine Arbeit mehr hat. Dieser sucht dann z. B. bei anderen Prozessoren nach freien Tasks und führt sie aus (Stehlen von Tasks). Senderbasierte Taskpools versuchen dagegen, neu erzeugte Tasks bei der Eingabe in den Taskpool so zu plazieren, daß alle Prozessoren immer Arbeit vorfinden. Für die Realisierung von Taskpools auf SMP-Clustern benutzen Hippold u. Rünger (2006) spezielle Kommunikationsthreads, um die Kommunikation von den Arbeitsthreads zu trennen. Eine Anzahl verschiedener Implementierungsmöglichkeiten für Taskpools, die unterschiedliche Basisdatenstrukturen, Lastbalancierungsstrategien und Synchronisationsmechanismen verwenden, wurden von Korch u. Rauber (2002, 2004b) und Hoffmann u. a. (2004a, b) implementiert und untersucht.

Einige anwendungsunabhängige Ansätze zur Steuerung der parallelen Abarbeitung irregulärer Probleme wurden in Form von Programmiersprachen und Bibliotheken zur Verfügung gestellt, z. B. Cilk (Blumofe u. a. 1995; Randall 1998; Supercomputing Technologies Group 2001), Charm++ (Kale u. Krishnan 1996), DOTS (Blochinger u. a. 1998; Blochinger u. Küchlin 2003), VDS (Decker 2000), Linda (Carriero u. Gelernter 1989) und DLBL (Carino u. Banicescu 2005).

3.6.3 Realisierung von Lastbalancierungsstrategien für eingebettete Runge-Kutta-Verfahren

Die Realisierung eines Lastbalancierungsverfahrens erfordert im allgemeinen die Beantwortung folgender Fragen:

1. Welche Programmteile können als Arbeitseinheiten (Tasks) verwendet werden?
2. Wie realisiert man die Arbeitseinheiten, damit die resultierende Granularität einerseits groß genug ist, um einen geringen Overhead der Lastbalancierung zu ermöglichen, aber andererseits auch hinreichend klein ist, um eine möglichst gleichmäßige Verteilung der Last zu gewährleisten?
3. Welche Anfangsverteilung der Arbeitseinheiten auf die teilnehmenden Kontrollflüsse verwendet man?
4. Wann und nach welcher Strategie sollen Arbeitseinheiten zwischen Kontrollflüssen migriert werden?
5. Wie kann man die Migration der Arbeitseinheiten anhand dieser Strategie realisieren? Welche Datenstrukturen und Synchronisationsmechanismen sind dazu notwendig?

Wahl und Realisierung der Arbeitseinheiten

Für eingebettete Runge-Kutta-Verfahren, die die Parallelität bezüglich des Differentialgleichungssystems ausnutzen, führt die Realisierung einer Lastbalancierungsstrategie zu einem dynamischen Scheduling-Problem für die parallelen Schleifen über die Systemdimension. Es ist daher naheliegend, die Iterationen der Schleifen, die jeweils eine Komponente des Differentialgleichungssystems verarbeiten, als Arbeitseinheiten zu verwenden.

Um abschätzen zu können, ob dies zu einer geeigneten Taskgranularität führt, blicken wir auf die Analyse der Funktionsauswertungszeiten in Abschnitt 3.6.1 zurück. Die dort betrachteten Testprobleme besitzen für den in Abb. 3.15 dargestellten Fall zwischen 66 und 400 Komponenten. Die Anzahl der pro Funktionsauswertung ausgeführten Instruktionen liegt zwischen 42 und 4085; die Anzahl der dazu erforderlichen Taktzyklen liegt zwischen 42 und ca. 8000. Die Dimension der Testprobleme MEDAKZO und BRUSS2D, die aus PDEs abgeleitet wurden, wächst quadratisch, wenn das zur Ortsdiskretisierung verwendete Gitter verfeinert wird. Die Funktionsauswertungszeiten der Komponenten ändern sich dabei nur unwesentlich. Die Funktionsauswertungszeit der Komponenten des Testproblems STARS hängt dagegen linear von der Anzahl der betrachteten Sterne ab, mit deren Anzahl auch die Systemdimension linear wächst. Zwar handelt es sich bei den betrachteten Testproblemen nur um wenige Beispiele, sie vermitteln jedoch bereits einen Eindruck von dem weiten Spektrum möglicher Systemdimensionen und Funktionsauswertungszeiten. Aus diesem Grund kann die Frage, ob die Verwendung einzelner Schleifeniterationen als Arbeitseinheiten eine geeignete Granularität bietet, nicht allgemein beantwortet werden.

Aufgrund der existierenden Vielzahl von Differentialgleichungssystemen mit unterschiedlichen Eigenschaften ist es vielmehr erforderlich, die Taskgranularität an die Eigenschaften des zu lösenden Problems

anzupassen. Es lassen sich jedoch folgende allgemeinen Aussagen treffen, die bei der Wahl der Arbeitseinheiten beachtet werden sollten:

- Alle Lastbalancierungsverfahren erfordern einen gewissen Overhead gegenüber einer statischen Arbeitsverteilung. Für viele Lastbalancierungsverfahren wächst dieser Overhead mit der Anzahl der zu verarbeitenden Arbeitseinheiten.
- Für preiswerte Funktionsauswertungen, die z. B. weniger als 100 Instruktionen benötigen, ist die Anzahl der neben der Funktionsauswertung innerhalb einer Iteration ausgeführten Instruktionen mit der Anzahl der für die Funktionsauswertung erforderlichen Instruktionen in etwa vergleichbar, oder sie ist deutlich kleiner. Das genaue Verhältnis ist von der Schleifenstruktur der Implementierung abhängig.
- Im Fall preiswerter Funktionsauswertungen sind die Kosten einer einzelnen Iteration deshalb sehr gering, und es besteht die Gefahr, daß der für die Lastbalancierung pro Arbeitseinheit erforderliche Overhead die Laufzeit dominiert. Eine Aufsplittung einzelner Schleifeniterationen, um eine noch feinere Taskgranularität zu erzielen, wird daher in der Regel nicht sinnvoll sein. Vielmehr wird man in dieser Situation eine Verbesserung der Laufzeit oft dadurch erreichen können, daß man die Granularität durch Zusammenfassen mehrerer Schleifeniterationen zu einer Arbeitseinheit vergrößert.
- Werden mehrere Iterationen zu einer Arbeitseinheit zusammengefaßt, sollten dazu direkt aufeinanderfolgende Iterationen benutzt werden, um ein möglichst gutes Lokalitätsverhalten zu erreichen.
- Die Anzahl der Cache-Fehlzugriffe durch False Sharing kann dadurch reduziert werden, daß die Arbeitseinheiten so gewählt werden, daß sie mit ein oder mehreren Cachezeilen korrespondieren.
- Durch das Zusammenfassen mehrerer Iterationen zu einer Arbeitseinheit sinkt die Anzahl der Arbeitseinheiten. Für eine effiziente Arbeitsweise der Lastbalancierung muß sichergestellt werden, daß die Anzahl der Arbeitseinheiten deutlich höher als die Anzahl der verwendeten Prozessoren ist.
- Besteht eine Arbeitseinheit aus einer festen Anzahl von Iterationen, kann es zu einer nichtbehebbaren Lastungleichverteilung kommen, wenn die Systemgröße kein ganzzahliges Vielfaches der Größe der Arbeitseinheiten ist.
- Je größer die Funktionsauswertungszeiten sind, um so weniger Iterationen sollten zu einer Arbeitseinheit zusammengefaßt werden, da bereits aus den höheren Funktionsauswertungszeiten eine grobere Granularität resultiert.

Da für die Lastbalancierung eingebetteter Runge-Kutta-Verfahren Schleifeniterationen bzw. Gruppen aufeinanderfolgender Schleifeniterationen als Arbeitseinheiten verwendet werden, können die zugehörigen Tasks durch eine oder zwei ganze Zahlen repräsentiert werden. Bei Verwendung einzelner Schleifeniterationen als Tasks kann dies z. B. der Index der Schleifeniteration sein. Bei Verwendung von Gruppen, bestehend aus einer festen Anzahl von Iterationen, genügt der Index der ersten Iteration dieser Gruppe oder eine spezielle Gruppennummer. Soll die Größe der Gruppen variabel sein, müssen mindestens zwei ganze Zahlen zur Repräsentation der Tasks verwendet werden (z. B. der Index der ersten und der letzten zur Gruppe gehörigen Iteration oder der Index der ersten zur Gruppe gehörigen Iteration und die Anzahl der Iterationen innerhalb der Gruppe).

Datenstrukturen zur Speicherung der Tasks

Jedes Lastbalancierungsverfahren muß geeignete Datenstrukturen zur Speicherung der ausführbaren Tasks bereitstellen. Diese Datenstrukturen bzw. deren Basisdatenstrukturen bezeichnet man als *Warteschlangen*, da die in ihnen gespeicherten Tasks gewissermaßen auf ihre Ausführung warten. Abhängig von der Strategie, nach der Tasks aus den Warteschlangen entnommen bzw. von einer Warteschlange in eine andere migriert werden, müssen Warteschlangen bestimmte Operationen unterstützen, mit deren Hilfe Tasks in die Warteschlangen eingefügt und entnommen werden können. Warteschlangen lassen sich u. a. wie folgt klassifizieren:

1. Anhand der Zugriffsrechte:

- **private Warteschlangen:** Auf die Warteschlange wird nur durch einen einzigen Kontrollfluß zugegriffen.
- **öffentliche Warteschlangen:** Auf die Warteschlange können Zugriffe durch mehrere Kontrollflüsse erfolgen. Es muß deshalb durch Verwendung geeigneter Synchronisationsmechanismen sichergestellt werden, daß beim Zugriff auf die Warteschlange keine zeitkritischen Abläufe auftreten.

2. Anhand der Häufigkeit des Zugriffs durch unterschiedliche Kontrollflüsse:

   - **verteilte Warteschlangen:** Die Warteschlange wird ausschließlich oder vorrangig von einem einzelnen Kontrollfluß oder einer einzelnen Gruppe von Kontrollflüssen benutzt.
   - **zentrale Warteschlangen:** Die Warteschlange wird von allen beteiligten Kontrollflüssen bzw. von allen Kontrollflüssen innerhalb einer betrachteten Gruppe von Kontrollflüssen benutzt.

   Werden die Warteschlangen in Form einer mehrstufigen Hierarchie organisiert, sind die Warteschlangen auf den mittleren Hierarchieebenen beiden Klassen zugehörig, da sie bezüglich der übergeordneten Hierarchieebene als verteilte und bezüglich der darunterliegenden Hierarchieebene als zentrale Warteschlangen betrachtet werden können. Kontrollflüsse, die ausschließlich zur Lastbalancierung dienen (z. B. der Master bei Verwendung eines Master-Slave-Modells), werden bei dieser Klassifizierung nicht mit einbezogen.

3. Anhand der Reihenfolge, in der die in die Warteschlange eingefügten Tasks aus dieser wieder entnommen werden, z. B.:

   - **FIFO-Warteschlangen:** FIFO steht für englisch: *first in, first out*. Die Tasks werden in der Reihenfolge entnommen, in der sie eingefügt wurden.
   - **LIFO-Warteschlangen:** LIFO steht für englisch: *last in, first out*. Die Tasks werden in der umgekehrten Reihenfolge entnommen, in der sie eingefügt wurden.
   - **Prioritätswarteschlangen:** Jedem Task wird eine Priorität zugeordnet. Die Tasks werden in der durch die Prioritäten definierten Reihenfolge entnommen.

Da im Fall der eingebetteten Runge-Kutta-Verfahren Tasks durch ein oder zwei ganze Zahlen repräsentiert werden können und zum Zweck der Lastbalancierung lediglich die Iterationen von Schleifen mit festen Grenzen auf die teilnehmenden Kontrollflüsse verteilt werden müssen und keine zusätzlichen Tasks während der Ausführung erzeugt werden, können Warteschlangen für die Lastbalancierung dieser Verfahren mit sehr einfachen Basisdatenstrukturen realisiert werden.

Beispiel 3.1.

1. Es soll eine Lastbalancierungsstrategie basierend auf einer einzelnen zentralen Warteschlange realisiert werden, wobei die von 1 bis n numerierten Komponenten des Differentialgleichungssystems als Arbeitseinheiten verwendet und in aufsteigender Reihenfolge der Indizes verarbeitet werden sollen. Um dies zu realisieren, genügt ein einfacher Zähler, der zu Beginn mit 1 initialisiert wird. Zur Entnahme eines Tasks wird der Zähler ausgelesen und anschließend erhöht. Um zeitkritische Abläufe zu vermeiden, muß dieser Vorgang durch wechselseitigen Ausschluß geschützt werden (z. B. durch Sperren einer Mutex-Variable oder durch die Verwendung atomarer Operationen). Erreicht der Zähler den Wert $n + 1$, wurden alle Tasks aus der Warteschlange entnommen, d. h., alle Iterationen der parallelen Schleife wurden an die beteiligten Kontrollflüsse zur Ausführung zugewiesen.

2. Eine öffentliche FIFO-Warteschlange für Taskgruppen, bestehend aus einer festen Anzahl aufeinanderfolgender Iterationen, soll realisiert werden, wobei keine Annahme über die Reihenfolge, in der die Abarbeitung der Gruppen erfolgt, getroffen werden soll. Unter dieser Voraussetzung ist es erforderlich, daß die Nummern aller sich in der Warteschlange befindenden Gruppen gespeichert werden; ein einfacher Zähler genügt nicht. Nehmen wir an, daß die Gruppen von 1 bis n_G numeriert sind. Da mit n_G auch eine Obergrenze für die maximale Zahl von Tasks, die sich gleichzeitig in einer Warteschlange befinden können, bekannt ist, kann zur Implementierung der Warteschlange ein lineares Ganzzahl-Feld der festen Größe n_G benutzt werden. Zusätzlich werden zur Realisierung einer FIFO-Schlange lediglich zwei

Indexzeiger benötigt, die auf die Feldpositionen verweisen, wo sich der erste in der Schlange gespeicherte Task befindet bzw. wo der nächste einzufügende Task gespeichert werden soll. Da es sich um eine öffentliche Warteschlange handeln soll, müssen alle Zugriffe auf die Schlange durch wechselseitigen Ausschluß geschützt werden.

3. Wie im Beispiel zuvor soll eine öffentliche Warteschlange für Gruppen, bestehend aus einer festen Anzahl aufeinanderfolgender Iterationen, realisiert werden, wobei diesmal jedoch festgelegt wird, daß die Verarbeitung der Gruppen in nach aufsteigendem Index geordneter Reihenfolge durchgeführt werden soll. Zusätzlich soll es im Unterschied zum vorigen Beispiel möglich sein, Tasks von beiden Enden einer Warteschlange zu entnehmen. Durch die Verarbeitung der Gruppen in nach aufsteigendem Index geordneter Reihenfolge kann die Warteschlange ähnlich wie im ersten Beispiel durch zwei Zähler realisiert werden. Der zweite Zähler ist nötig, da aufgrund der Möglichkeit zur Entnahme von Tasks an beiden Enden der Index des letzten Elements der Warteschlange nicht festgelegt ist.

Diese Beispiele machen deutlich, daß die Antwort auf die Frage, welche Art, welche Organisation und welche Implementierung von Warteschlangen verwendet werden sollte, von den Anforderungen des zu realisierenden Lastbalancierungsverfahrens abhängig ist. Gelten bestimmte Einschränkungen, so kann man einfachere Basisdatenstrukturen mit geringeren Zugriffskosten verwenden. Ist jedoch eine komplexe Hierarchie von Warteschlangen mit flexiblen Zugriffsmöglichkeiten erforderlich, ergibt sich ein dementsprechend höherer Verwaltungsaufwand. Aus diesem Grund richtet sich die Auswahl geeigneter Datenstrukturen nicht zuletzt auch nach dem maximalen Overhead, der für die durch das Differentialgleichungssystem und die Wahl der Arbeitseinheiten gegebene Granularität zulässig ist. Da viele Differentialgleichungssysteme eine sehr feine Taskgranularität erfordern, wird man in der Regel einfache Datenstrukturen mit geringem Overhead bevorzugen.

Wahl der Anfangsverteilung

Für irreguläre Algorithmen mit einer sehr groben Taskgranularität ist es möglich, auf eine Anfangsverteilung der Arbeitseinheiten auf die teilnehmenden Kontrollflüsse zu verzichten, indem man zur Lastbalancierung eine einzelne zentrale Warteschlange verwendet. Denn obwohl in diesem Fall alle auf der Warteschlange ausgeführten Operationen in Konkurrenz zueinander stehen, ist die Wahrscheinlichkeit, daß sich Kontrollflüsse gegenseitig blockieren (z. B. beim Zugriff auf eine Mutex-Variable oder beim Transfer einer Nachricht über ein Verbindungsnetzwerk), aufgrund der groben Taskgranularität gering. Die für die Lastbalancierung eingebetteter Runge-Kutta-Verfahren erforderliche Taskgranularität ist jedoch in der Regel sehr fein, so daß die Verwendung einer einzelnen zentralen Warteschlange aufgrund der hohen Zugriffsfrequenz zu hohen Wartezeiten führen kann. Experimente mit anderen feingranularen irregulären Anwendungen (hierarchisches Radiosity-Verfahren und Volume Rendering, siehe Korch u. Rauber 2002, 2004b) belegen, daß in dieser Situation Lastbalancierungsstrategien, die ausschließlich oder zusätzlich zu zentralen Warteschlangen auch auf verteilten Warteschlangen beruhen, eine bessere Gesamtlaufzeit erreichen. Dies setzt voraus, daß zu Beginn eine geeignete Anfangsverteilung der Tasks erfolgt, die im Nachhinein nur noch wenige konkurrierende Zugriffe auf öffentliche Warteschlangen zum Zweck des Lastausgleichs erfordert.

Nun stellt sich jedoch die Frage, wie man eine solch günstige Anfangsverteilung bestimmen kann. Ohne Informationen über die Funktionsauswertungszeiten des zu integrierenden Differentialgleichungssystems und die Arbeitsgeschwindigkeit der beteiligten Kontrollflüsse ist dies äußerst schwierig. Man kann sich jedoch mit der Annahme behelfen, daß alle Arbeitseinheiten etwa die gleiche Berechnungszeit erfordern, unabhängig davon, von welchem Kontrollfluß die Berechnung ausgeführt wird. Unter dieser Annahme wäre jede Anfangsverteilung optimal, bei der jeder Kontrollfluß die gleiche Anzahl von Arbeitseinheiten zugewiesen bekommt. Das bedeutet insbesondere, daß unter dieser Annahme die blockweise Verteilung, die auch in den Implementierungen ohne Lastbalancierung verwendet wird, optimal ist. Die Verwendung der gleichen Arbeitsverteilung wie in den Implementierungen ohne Lastbalancierung hat außerdem den Vorteil, daß auf keinen Fall eine schlechtere Ausgangssituation entsteht, als sie im Fall der Implementierungen ohne Lastbalancierung vorliegt. Ein weiterer entscheidender Vorteil der blockweisen Verteilung ist, daß sie aufgrund der Tatsache, daß alle Kontrollflüsse zusammenhängende Blöcke verarbeiten, eine besonders hohe Lokalität besitzt.

Hat man Grund zu der Annahme, daß für das zu lösende Problem eine blockweise Verteilung zu einem starken Lastungleichgewicht führen würde (z. B. im Fall von STARS-CON), sollte man andere Verteilungen, z. B. eine zyklische oder eine blockzyklische Verteilung, in Erwägung ziehen. Für bestimmte Probleme kann aber gerade eine solche Verteilung ungünstig sein (z. B. eine zyklische komponentenweise Verteilung auf zwei Kontrollflüsse für STARS-MIX). Kennt man die Struktur des Differentialgleichungssystems nicht und will man dennoch diese ungünstigen Fälle weitestgehend ausschließen, kann man auf randomisierte Verteilungen zurückgreifen. Da sich aber auch bei einer randomisierten Verteilung eine ungünstige Zuordnung ergeben kann, sollte die Verteilung im Verlauf der Integration mehrfach neu berechnet werden, z. B. nach einer bestimmten Anzahl von Zeitschritten oder in jeder Stufe. Dabei sollte man jedoch beachten, daß für die Bestimmung der erforderlichen Zufallszahlen ein im Vergleich zu den Kosten der Funktionsauswertungen relativ hoher Aufwand erforderlich sein kann. Alle soeben genannten Verteilungen besitzen darüber hinaus eine schlechtere Lokalität als die blockweise Verteilung.

Werden die Kontrollflüsse auf Recheneinheiten mit unterschiedlicher Arbeitsgeschwindigkeit ausgeführt, ist es sinnvoll, die Geschwindigkeiten der Recheneinheiten in die Bestimmung der Anfangsverteilung einzubeziehen. Sind diese zur Übersetzungszeit noch nicht bekannt, kann man sie auch zur Laufzeit vor Beginn der Integration bestimmen.

Die beste Möglichkeit zur Bestimmung einer geeigneten Anfangsverteilung besteht, wenn die Funktionsauswertungszeiten der ODE-Komponenten bekannt sind. Sind die Funktionsauswertungszeiten für $f_j(t, \mathbf{w})$ nicht von der Zeit t und dem Argumentvektor $\mathbf{w}$ abhängig, kann man diese, wie dies z. B. in Abschnitt 3.6.1 erfolgt ist, mit Hilfe eines separaten Programms bestimmen und dem Integrator als zusätzliche Eingabe zur Verfügung stellen. Im allgemeinen kann es jedoch vorkommen, daß sich die Funktionsauswertungszeiten im Verlauf der Integration verändern, beispielsweise bei der Simulation einer chemischen Reaktion, bei der nach Ablauf einer bestimmten Zeitspanne oder bei Erreichen eines bestimmten Zustands eine weitere Substanz hinzugegeben wird. In diesem Fall müssen die Funktionsauswertungszeiten zur Laufzeit der Integration mehrfach neu bestimmt werden. Dazu kann man die Laufzeit der Funktionsauswertungen entweder durch Abfrage des Systemzeitgebers oder durch das Auswerten von Hardware-Performance-Countern messen. Anhand der Funktionsauswertungszeiten kann dann eine ungleichförmige blockweise Verteilung berechnet werden, welche die Arbeitseinheiten so an die beteiligten Kontrollflüsse verteilt, daß alle Kontrollflüsse zur Verarbeitung der ihnen zugewiesenen Arbeitseinheiten etwa die gleiche Zeit benötigen. Die Verwendung einer ungleichförmigen blockweisen Verteilung hat gegenüber anderen Verteilungen den Vorteil einer hohen Lokalität.

Die Bestimmung der Funktionsauswertungszeiten und die Adaption der Arbeitsverteilung zur Laufzeit der Integration erfordert allerdings einen gewissen Overhead. Es lohnt sich daher nur, diesen Zeitaufwand zu investieren, wenn er durch den erzielten Laufzeitgewinn übertroffen wird. Deshalb muß man versuchen, den Overhead zur Adaption der Datenverteilung so gering wie möglich zu halten. Eine Möglichkeit dazu ist die Ausnutzung von Parallelität, indem die neue Verteilung der Arbeitseinheiten parallel zur Schrittweitenkontrolle ausgeführt wird. Da die Schrittweitenkontrolle einen sequentiellen Anteil jedes Zeitschrittes bildet, kann so zumindest ein Teil der für die Anpassung der Datenverteilung notwendigen Instruktionen ohne zusätzlichen Zeitaufwand ausgeführt werden. In bestimmten Fällen lassen sich auch Wartezeiten ausnutzen, die durch die Ausführung von Synchronisationsoperationen (z. B. Barrier-Operationen oder Multibroadcastoperationen zwischen den Stufen) entstehen.

Strategien zur Migration von Tasks

Die Migration von Tasks, d. h. das Verschieben von Tasks aus einer Warteschlange in eine andere und somit die Veränderung der Zuordnung zu einem bestimmten Prozessor oder einer Prozessorengruppe, ist notwendig, um zur Laufzeit einen Lastausgleich herbeiführen zu können. Da die für die Taskmigration benötigte Zeit in die Laufzeit mit einfließt, ohne daß während dieser Zeit Berechnungen ausgeführt werden, die direkt zur Lösung des Problems beitragen, sollte der Gesamtaufwand für die Taskmigration im Verhältnis zur Gesamtlaufzeit möglichst klein sein. Um dies zu erreichen, sollten

1. die Anzahl der Migrationen und

2. die Kosten pro Migration

minimiert werden. Zusätzlich zu den eigentlichen Migrationskosten entsteht ein Overhead

3. zur Ermittlung der aktuellen Lastverteilung bzw. für das Treffen der Entscheidung, ob eine Migration von Tasks erforderlich ist, und

4. zur Gewährleistung des wechselseitigen Ausschlusses beim Zugriff auf öffentliche Warteschlangen.

Dieser Overhead tritt z. T. auch dann auf, wenn keine Migration ausgeführt werden muß. Er übt deshalb auch bei einer optimalen Anfangsverteilung einen negativen Einfluß auf die Laufzeit aus. Dies bedeutet insbesondere, daß man bei einer guten Anfangsverteilung, die nur wenige Taskmigrationen erfordert, höhere Migrationskosten in Kauf nehmen kann, wenn es dadurch möglich ist, den von der Anzahl der Migrationen unabhängigen Overhead zu senken.

Strategien zur Migration von Tasks, bei denen verteilte Warteschlangen zum Einsatz kommen, lassen sich u. a. anhand des Kontrollflusses klassifizieren, der die Migration initiiert:

- **Senderbasierte Strategien:** Migrationen werden von taskverarbeitenden Kontrollflüssen initiiert, die Tasks abgeben möchten.
- **Empfängerbasierte Strategien:** Migrationen werden von taskverarbeitenden Kontrollflüssen initiiert, die Tasks erhalten möchten.
- **Entscheiderbasierte Strategien:** Ein spezieller Kontrollfluß, der selbst keine Tasks verarbeitet, ist dafür verantwortlich, die Lastverteilung zu überwachen und gegebenenfalls Migrationen zu initiieren.
- **Gemischte Strategien:** Mehrere der zuvor genannten Strategien werden miteinander kombiniert.

Senderbasierte Strategien kommen vor allem dann in Betracht, wenn zur Ausführungszeit neue Tasks erzeugt werden. In diesem Fall bietet es sich an, daß ein Kontrollfluß, der gerade einen Task erzeugt hat, seine Last im Vergleich zu den übrigen Kontrollflüssen einschätzt und gegebenenfalls den neu erzeugten Task oder auch weitere Tasks aus einer ihm zugeordneten Schlange in eine andere Schlange migriert. Kommen jedoch keine neuen Tasks während der Ausführung hinzu, hat eine senderbasierte Strategie den Nachteil, daß für die Entscheidung, ob die Last des Senders hoch genug ist, um ein Versenden von Tasks zu rechtfertigen, die Last anderer Kontrollflüsse ermittelt werden muß, um diese zum Vergleich heranziehen zu können. Bei einer empfängerbasierten Strategie genügt es dagegen, festzustellen, daß die Last des Empfängers niedrig genug ist, um eine Migration zu rechtfertigen. Ein sehr einfaches Kriterium dafür ist beispielsweise, daß der Empfänger keine Tasks mehr in einer ihm zugeordneten Warteschlange vorfindet. Bei Verwendung eines derartigen Kriteriums ist eine Betrachtung der Last anderer Kontrollflüsse nicht erforderlich, solange die Last des potentiellen Empfängers noch ausreichend hoch ist.

Bei Verwendung einer entscheiderbasierten Strategie wird der Programmcode zur Steuerung der Lastverteilung in einen oder mehrere separate Kontrollflüsse ausgelagert. Dies ist sinnvoll, wenn CPU-Ressourcen zur Verfügung stehen, die durch diese zusätzlichen Kontrollflüsse ausgenutzt werden können und andernfalls ungenutzt blieben. Beispielsweise könnte eine Rechnerarchitektur eine zusätzliche Master-CPU mit spezialisiertem Funktionsumfang zur Verfügung stellen. Aber auch der Einsatz einer zusätzlichen regulären CPU kann vorteilhaft sein, wenn z. B. das Entfernen des Lastbalancierungscodes aus den Arbeitskontrollflüssen deren jeweilige Effizienz so stark verbessert, daß sich die parallele Effizienz des Gesamtprogramms trotz Einsatz der zusätzlichen CPU ebenfalls erhöht. Weiterhin kann die Ausführung der Lastbalancierungskontrollflüsse auf den gleichen CPUs wie die Arbeitskontrollflüsse zu einer Laufzeitverbesserung führen, wenn es dadurch gelingt, die Auslastung der CPUs zu verbessern, indem Wartezeiten der Arbeitskontrollflüsse zur Lastbalancierung genutzt werden.

Die Lastbalancierung eingebetteter Runge-Kutta-Verfahren erfordert Migrationsstrategien, die auf die typischerweise sehr feine Taskgranularität abgestimmt sind. Diese müssen sich vor allem durch einen insgesamt sehr geringen Overhead auszeichnen. Da der Einsatz von Lastbalancierungsverfahren u. a. dadurch motiviert wurde, daß auch geringe Laufzeitunterschiede zwischen den Kontrollflüssen ausgeglichen werden können, die z. B. durch unterschiedliche Speicherzugriffszeiten und durch Unterbrechungen durch das Betriebssystem hervorgerufen werden, muß der Overhead geringer sein als diese verhältnismäßig geringen Laufzeitunterschiede zwischen den Kontrollflüssen. Das bedeutet im einzelnen:

- Es sollte eine möglichst gute Anfangsverteilung gewählt werden, so daß zum Lastausgleich nur noch wenige Migrationen erforderlich sind (siehe vorangehender Abschnitt).

- Obwohl bei einer guten Anfangsverteilung nur wenige Migrationen ausgeführt werden müssen, dürfen die Migrationskosten ebenfalls nicht zu groß sein.
- Wenn eine Migration durchgeführt wird, sollten Quell- und Zielwarteschlange sowie die Anzahl der zu migrierenden Tasks so gewählt werden, daß die Anzahl nachfolgend erforderlicher Migrationen möglichst reduziert wird.
- Der zur Taskverarbeitung erforderliche Overhead zur Entnahme der Tasks aus einer Warteschlange und zur Überprüfung der Lastverteilung sollte so klein wie möglich sein.
- Kritische Programmbereiche, die unter wechselseitigem Ausschluß ausgeführt werden, dürfen nur sehr kurz sein, um andere Kontrollflüsse nicht unnötig zu blockieren. Um Wartezeiten zu vermeiden, kann es sinnvoll sein, daß Kontrollflüsse, die einen kritischen Bereich nicht betreten können, zwischenzeitlich andere Berechnungen ausführen oder auf andere kritische Bereiche ausweichen.

Diese Forderungen verlangen nach Datenstrukturen zur Speicherung der Tasks, die mit wenigen Operationen manipulierbar sind. Sie setzen weiterhin eine parallele Rechnerarchitektur basierend auf einem schnellen Verbindungsnetzwerk mit geringen Latenzzeiten voraus, die effiziente Synchronisationsmechanismen zur Verfügung stellt. Für Anwendungen mit einer sehr feinen Taskgranularität ist in der Regel die Verwendung von verteilten Warteschlangen vorteilhaft, was auch Skalierbarkeitsuntersuchungen von Lastbalancierungsverfahren für andere feingranulare Anwendungen (Korch u. Rauber 2002, 2004b) belegen. Speziell die Forderung nach einer effizienten Realisierung der Taskentnahme und der Überprüfung der Lastverteilung legt die Verwendung einer empfängerbasierten Strategie nahe, die allein anhand der dem potentiellen Empfänger zugeordneten Last entscheidet, ob eine Migration von Tasks erforderlich ist. Eine einfache, aber sehr effektive Strategie besteht z. B. darin, daß jeder Arbeitskontrollfluß bei der Entnahme eines Tasks aus einer ihm zugeordneten Warteschlange die Anzahl der in dieser Warteschlange gespeicherten Tasks überprüft. Unterschreitet diese einen bestimmten Schwellwert, werden Tasks von einem anderen Kontrollfluß „gestohlen“, d. h. aus dessen Warteschlange in die Warteschlange des Empfängers transferiert. Um die Anzahl der Tasktransfers zu reduzieren, kann dabei gezielt eine größere Anzahl von Tasks migriert werden.

Da der Zugriff auf die Warteschlange des Senders in der Regel unter wechselseitigem Ausschluß erfolgen muß, kann während des Stehlens kein anderer Kontrollfluß einschließlich des Senders selbst Tasks aus der Warteschlange des Senders entnehmen. Aus diesem Grund sollte das Stehlen so schnell wie möglich abgeschlossen werden, um Wartezeiten der anderen Kontrollflüsse weitestgehend zu minimieren. Die zur Realisierung der Warteschlangen verwendeten Basisdatenstrukturen müssen deshalb so aufgebaut sein, daß nur wenige Operationen zur Entnahme von Tasks ausgeführt werden müssen. Um effizient mehrere Tasks stehlen zu können, sollten die Kosten zur Entnahme mehrerer Tasks im Idealfall unabhängig von der Anzahl der Tasks sein. Ein Beispiel für eine Datenstruktur, die dies ermöglicht, ist die bereits im Abschnitt über Datenstrukturen zur Speicherung von Tasks angesprochene, mit Hilfe von zwei Zählern realisierte Warteschlange zur Speicherung einer zusammenhängenden Gruppe von Tasks mit aufeinanderfolgenden Tasknummern.

Realisierung des wechselseitigen Ausschlusses

Da für eingebettete Runge-Kutta-Verfahren die Ausführungszeit eines Tasks z. T. unter 100 Taktzyklen betragen kann, führt oft bereits ein Overhead von nur 5 bis 10 Takten pro Taskentnahme zu einer inakzeptablen Vergrößerung der Laufzeit. Insbesondere dann, wenn bereits eine nahezu ausgeglichene Anfangsverteilung gegeben ist, kann eine solche 5- bis 10-prozentige Verlangsamung nicht durch eine verbesserte Lastverteilung aufgewogen werden. Zur Realisierung des wechselseitigen Ausschlusses beim Zugriff auf Warteschlangen sollten deshalb möglichst effiziente Synchronisationsmechanismen verwendet werden, die pro Taskentnahme nur einen sehr geringen Overhead von wenigen Takten benötigen.

Auf parallelen Rechnersystemen mit einem verteilten Adreßraum ist es in der Regel schwierig, Synchronisationsmechanismen mit einem derart geringen Overhead zu realisieren. Zwar besitzen einige moderne Rechnersysteme z. T. sehr schnelle Verbindungsnetzwerke, die innerhalb der physikalischen Schicht geringe Latenzzeiten erreichen, jedoch erfolgt die Benutzung des Verbindungsnetzwerks oft mit Hilfe spezieller

Treiber und Kommunikationsbibliotheken (z. B. MPI), so daß die übertragenen Nachrichten oft eine Vielzahl von Softwareschichten durchlaufen. Entsprechend hoch sind die insgesamt resultierenden Latenzzeiten. Doch selbst wenn man die Kommunikationshardware direkt aus dem Anwendungsprogramm heraus ansprechen würde und zusätzlich versucht, die Kommunikation weitestgehend parallel zu anderen Berechnungen auszuführen (z. B. durch Nutzung von DMA-Transfers), dürfte der verbleibende Synchronisationsoverhead auf den meisten Systemen immer noch zu groß für eine effiziente Lastbalancierung feingranularer Anwendungen sein.

Beispiel 3.2. Gegeben sei ein paralleles Rechnersystem mit verteiltem Adreßraum. Jeder Knoten des Rechnersystems besitzt einen Netzwerk-Controller, dessen I/O-Bereiche auf Speicheradressen des Knotens abgebildet sind. Realisiert werden soll eine auf einem einfachen Zähler basierende verteilte Warteschlange. Die Entnahme eines Tasks bedeutet also, den Zähler zu lesen und den um eins erhöhten Wert zurückzuschreiben. Unter diesen Voraussetzungen kann der Zugriff auf die Warteschlange z. B. wie folgt realisiert werden:

1. Jeder Knoten entnimmt Tasks aus seiner lokalen Warteschlange, in dem er den Wert des Zählers liest, um eins erhöht und den erhöhten Wert zurückschreibt. Zur Gewährleistung des wechselseitigen Ausschlusses wird während der Taskentnahme der Interrupt des Netzwerk-Controllers deaktiviert.

2. Ein Zielknoten, der einen Task aus der lokalen Warteschlange eines Quellknotens entnehmen möchte, muß eine Nachricht an den Quellknoten schicken. Die Nachricht muß lediglich ein Kommando enthalten, das dem Quellknoten die Art der Anfrage mitteilt, sowie die Adresse des Zielknotens. Diese Nachricht wird durch Speicheroperationen auf dem I/O-Bereich des Netzwerk-Controllers des Zielknotens zusammengestellt und abgesandt.

3. Der Netzwerk-Controller des Quellknotens empfängt die Nachricht und löst einen Interrupt aus. Der Interrupt-Handler liest und erhöht den Wert des Zählers. Anschließend wird durch die Ausführung von Speicheroperationen auf dem I/O-Bereich des Netzwerk-Controllers eine Nachricht mit dem ursprünglich gelesenen Wert des Zählers zusammengestellt und an den Zielknoten gesendet.

4. Der Netzwerk-Controller des Zielknotens empfängt die Antwort des Quellknotens und löst einen Interrupt aus. Der Interrupt-Handler übergibt die Nummer des Tasks an den Arbeitskontrollfluß, der diesen daraufhin verarbeitet.

Auf Systemen mit gemeinsamem Adreßraum erfolgt der Datenaustausch zwischen den Kontrollflüssen dagegen durch gemeinsame Variablen, die direkt über Speicherzugriffsoperationen adressiert werden können. Zwar müssen auch hier die zu übertragenden Daten ein Verbindungsnetzwerk durchlaufen, das Routing erfolgt jedoch allein durch Hardwaremechanismen anhand der zugegriffenen Adresse. Eine Beteiligung anderer CPUs ist nicht erforderlich. Um wechselseitigen Ausschluß bei der Manipulation von Speicherzellen zu ermöglichen, werden spezielle, nichtunterbrechbare Maschinenbefehle zur Verfügung gestellt. Oft sind dies sogenannte Read-Modify-Write-Operationen, die das atomare Auslesen, Verändern und Zurückschreiben eines Datums ermöglichen (z. B. Fetch & Add oder Compare & Swap). Ein anderer Ansatz besteht darin, spezielle Lese- und Schreiboperationen (z. B. Load & Reserve und Store Conditional) zu realisieren, die beim Lesen eines Datums eine Reservierung setzen und das zwischenzeitlich vom Benutzer modifizierte Datum nur unter der Voraussetzung zurückschreiben, daß zuvor kein anderer Schreibzugriff auf dieses Datum die Reservierung entfernt hat. Auf Basis solcher atomarer Operationen können flexiblere, aber gleichzeitig auch komplexere Synchronisationsmechanismen wie z. B. Spin-Locks, passiv wartende Locks, Bedingungsvariablen oder Barrier-Operationen realisiert werden. Solch komplexe Synchronisationsmechanismen werden in der Regel durch standardisierte Anwendungsschnittstellen (z. B. POSIX Threads und OpenMP) zur Verfügung gestellt.

Zwar sind die von standardisierten Anwendungsschnittstellen zur Verfügung gestellten Synchronisationsmechanismen einfach zu benutzen und flexibel einsetzbar, jedoch erzeugt ihr Einsatz oft einen zu großen Overhead. Es kann deshalb für feingranulare Anwendungen vorteilhaft sein, zur Reduktion des Overheads Synchronisationsmechanismen in die Anwendung zu integrieren und für die speziellen Anforderungen der Anwendung zu optimieren bzw. spezielle anwendungsspezifische Synchronisationsmechanismen zu entwickeln. So haben beispielsweise von Hoffmann (2003) und Hoffmann u. a. (2004a, b) durchgeführte Untersuchungen einiger irregulärer taskbasierter Anwendungen gezeigt, daß sich Laufzeit

und Skalierbarkeit dieser Anwendungen durch die Integration auf Basis atomarer Operationen realisierter Spin-Locks sowie durch die Anwendung lock-freier Synchronisationsmechanismen signifikant verbessern lassen. Da die Taskgranularität eingebetteter Runge-Kutta-Verfahren in der Regel noch feiner ist als die der in den genannten Arbeiten untersuchten Anwendungen, ist deshalb davon auszugehen, daß eine effiziente Lastbalancierung eingebetteter Runge-Kutta-Verfahren ebenfalls die Integration der Synchronisationsmechanismen in den Programmcode des Lösungsverfahrens und in vielen Fällen sogar die Entwicklung anwendungsspezifischer Synchronisationsmechanismen erfordert.

Beispiel 3.3. Auf einem parallelen Rechnersystem mit gemeinsamem Adreßraum, das die atomare Operation Fetch & Add zur Verfügung stellt, soll analog Beispiel 3.2 eine verteilte, auf einem Zähler basierende Warteschlange realisiert werden. Die Kontrollflüsse können auf einem solchen System z. B. durch Threads realisiert werden. Jeder Thread verwaltet dann einen Zähler, der die Nummer des nächsten zu verarbeitenden Tasks enthält. Zur Entnahme eines Tasks aus einer Warteschlange genügt es, den betreffenden Zähler mit Hilfe der Fetch & Add Operation atomar auszulesen und um eins zu erhöhen.

3.6.4 Untersuchung des Laufzeitverhaltens verschiedener Lastbalancierungsstrategien

Im folgenden Abschnitt werden mehrere Realisierungen von Lastbalancierungsstrategien für eingebettete Runge-Kutta-Verfahren vorgestellt und deren Laufzeitverhalten im Vergleich zu einer statischen Arbeitsverteilung untersucht.

Betrachtete Lastbalancierungsstrategien

Um zu untersuchen, ob in der Praxis eine Laufzeitverbesserung durch die Verwendung von Lastbalancierungstechniken möglich ist, werden drei unterschiedliche Realisierungen von empfängerbasierten Lastbalancierungsstrategien für Rechnersysteme mit einem gemeinsamen Adreßraum betrachtet. Jede Lastbalancierungsstrategie wurde in zwei Varianten implementiert, die unterschiedliche Arbeitseinheiten unterstützen: Die erste der beiden Varianten verarbeitet einzelne Komponenten bzw. Schleifeniterationen, während die zweite Variante so viele Komponenten zu einer Arbeitseinheit zusammenfaßt, wie in einer Cachezeile der jeweiligen Zielarchitektur gespeichert werden können. Um die Anzahl der Cache-Fehlzugriffe zu reduzieren, die daraus resultieren, daß die zu einer Arbeitseinheit gehörenden Vektorelemente gleichzeitig in den Caches mehrerer Prozessoren gespeichert sind, werden in den auf Cachezeilenbasis arbeitenden Implementierungsvarianten die verwendeten Vektoren durch Padding so im Speicher ausgerichtet, daß die zu einer Arbeitseinheit gehörenden Elemente eines Vektors immer der gleichen Cachezeile angehören. Alle betrachteten Strategien verwenden als Anfangsverteilung der Tasks die gleiche blockweise Verteilung, die auch von den Implementierungen mit statischer Arbeitsverteilung benutzt wird. Die Lastbalancierungsimplementierungen wurden ausgehend von Implementierung (D) (Abb. 3.10) entwickelt, die auch als Referenz für den Laufzeitvergleich und die Speedup-Berechnung verwendet wird. Zur Implementierung der Kontrollflüsse werden POSIX Threads verwendet. Ein Überblick über alle betrachteten Varianten der Lastbalancierungsstrategien gibt Tab. 3.5.

Strategie „Simple“. Diese Strategie wurde mit dem Ziel implementiert, die Lastbalancierung mit einem möglichst geringen Instruktionsoverhead zu realisieren. Dazu wird ähnlich wie in Beispiel 3.3 eine verteilte Warteschlange mit Hilfe einfacher Zähler implementiert. Jeder Thread verwaltet einen Zähler, der auf den nächsten zu verarbeitenden Task verweist. Zur Berechnung einer Stufe werden diese Zähler zunächst mit der Nummer des ersten dem jeweiligen Thread zugeordneten Tasks initialisiert. Die Berechnung der Stufe erfolgt dann, indem jeder Thread durch atomares Auslesen und Erhöhen des Zählers solange Tasks aus der ihm zugeordneten Warteschlange entnimmt, bis der Zähler die Nummer des letzten ihm zugeordneten Tasks überschreitet. Tritt der Fall ein, daß ein Thread alle Tasks aus der ihm zugeordneten Warteschlange abgearbeitet hat, wechselt er entsprechend einer vorgegebenen Strategie seine Thread-ID und beginnt damit, Tasks aus der Warteschlange zu verarbeiten, die seiner neuen Thread-ID zugeordnet ist. Dieses Vorgehen wird fortgesetzt, bis jeder Thread jede Warteschlange genau einmal bearbeitet hat. Der sich daraus ergebende Algorithmus ist in Abb. 3.16 dargestellt.

Implementierung	Strategie	Arbeitseinheiten	Synchronisationsmechanismus
SCIL	Simple/Increment	Komponenten	Mutex-Variable
SPIL	Simple/Increment	Cachezeilen	Mutex-Variable
SCIA	Simple/Increment	Komponenten	atomares Fetch & Inc
SPIA	Simple/Increment	Cachezeilen	atomares Fetch & Inc
SCRA	Simple/Random	Komponenten	atomares Fetch & Inc
SPRA	Simple/Random	Cachezeilen	atomares Fetch & Inc
TC	Task Queue	Komponenten	Mutex-Variable
TP	Task Queue	Cachezeilen	Mutex-Variable
IC	Interval Queue	Komponenten	Mutex-Variable
IP	Interval Queue	Cachezeilen	Mutex-Variable

Tab. 3.5: Überblick über die implementierten Lastbalancierungsstrategien.

```
1:  for (l := 1; l ≤ s; l++) next_work_unit_l[me] := first_work_unit[me];
2:  current_id := me;

3:  for (l := 1; l ≤ s; l++)
4:  {
5:      barrier();

6:      loop
7:      {
8:          work_unit := FETCH_AND_INC(next_work_unit_l[current_id]);
9:          if (work_unit > last_work_unit[current_id])
10:         {
11:             current_id := NEXT_THREAD_ID(current_id);
12:             if (current_id = me) break;
13:         }
14:         else
15:             PROCESS_WORK_UNIT(l, work_unit);
16:     }
17: }
```

Abb. 3.16: Lastbalancierungsstrategie „Simple“.

Es wurden zwei unterschiedliche Strategien implementiert, anhand welcher ein Thread seine Thread-ID wechselt:

- „Increment“: Die ID wird um eins erhöht. Übersteigt sie danach die höchstmögliche ID, wird die kleinstmögliche ID zugewiesen. Diese Strategie wurde z. B. auch von Nieh u. Levoy (1992) zur Lastbalancierung eines Volume-Rendering-Verfahrens verwendet.

  Ein Nachteil dieser Strategie ist, daß mit relativ hoher Wahrscheinlichkeit der Fall eintreten kann, daß die Zugriffshäufigkeit auf nichtleere Warteschlangen ungleichmäßig verteilt ist. Betrachten wir als Beispiel die Integration des Testproblems STARS-CON mit vier Threads. Da die Komponenten $1, \ldots, n/2$ sehr schnell verarbeitet werden können, erhöhen die Threads 1 und 2 ihre Thread-ID bis auf 3, noch bevor Thread 3 mit der Bearbeitung der ihm zugeordneten Komponenten fertig ist. Infolgedessen greifen ab diesem Zeitpunkt 3 Threads auf die Warteschlange mit der ID 3 zu, während nur 1 Thread auf die Warteschlange mit der ID 4 zugreift. Die Warteschlange mit der ID 3 wird deshalb schneller abgearbeitet werden können als die Warteschlange mit der ID 4, so daß es letztendlich dazu kommt, daß alle vier Threads parallel die verbleibenden Threads aus der Warteschlange mit der ID 4 verarbeiten.

- „Random“: Aus diesem Grund wurde eine zweite Strategie zum Wechsel der Thread-ID implementiert, die durch eine Randomisierung der Reihenfolge der von den Threads angenommenen IDs eine solch ungünstige Verteilung der Zugriffshäufigkeiten in der Mehrzahl der Fälle vermeiden soll. Da die Berechnung von Zufallszahlen jedoch die Ausführung zusätzlicher Operationen erfordert, wird die Reihenfolge der IDs für jeden Thread vor Beginn der Integration festgelegt und während der Integration nicht mehr verändert, um eine Vergrößerung des Overheads durch die Zufallszahlenberechnung zu vermeiden.

Zur Entnahme eines Tasks aus einer Warteschlange muß der entsprechende Zähler gelesen, inkrementiert und zurückgeschrieben werden. Um zeitkritische Abläufe zu verhindern, muß ausgeschlossen werden, daß der betreffende Thread während der Ausführung dieser Operationen unterbrochen werden kann. Dies wurde in den untersuchten Implementierungen auf zwei verschiedene Arten realisiert. Der erste Ansatz besteht darin, von der POSIX-Thread-Bibliothek zur Verfügung gestellte Mutex-Variablen zu verwenden. Falls verfügbar, werden dazu Mutex-Variablen vom Typ `pthread_spinlock_t` verwendet. Andernfalls wird auf den Typ `pthread_mutex_t` zurückgegriffen. Der zweite Ansatz besteht in der Verwendung atomarer Maschinenoperationen zur Realisierung des Primitivs Fetch & Inc. Auf Rechnersystemen, auf denen eine solche Operation nicht als Maschinenbefehl zur Verfügung steht, wird sie durch andere Synchronisationsmechanismen emuliert. Von den derzeitigen Versionen der Implementierungen werden folgende Zielplattformen unterstützt:

- IA64: Es wird die vom IA64-Befehlssatz unterstützte Operation Fetch & Add verwendet.

- x86: Fetch & Inc wird mit Hilfe des Maschinenbefehls Compare & Swap emuliert, indem in einer Schleife der Wert des Zählers in ein Register geladen, um eins erhöht und mit Compare & Swap unter der Bedingung zurückgeschrieben wird, daß sich der Wert des Zählers im Speicher nicht verändert hat.

- AIX: Es wird die vom Kernel des Betriebssystems zur Verfügung gestellte Funktion `fetch_and_add()` genutzt.

Strategie „Task Queue“. Diese Strategie verwendet eine verteilte Warteschlange auf Basis einer komplexeren Datenstruktur, die ein hohes Maß an Flexibilität bezüglich der Reihenfolge der Verarbeitung der Tasks bietet. Als Basisdatenstruktur wird dazu pro Thread ein eindimensionales Feld benutzt, dessen Elemente die Nummern der in der Warteschlange gespeicherten Tasks enthalten. Da die maximale Zahl der in einer Warteschlange gespeicherten Tasks bekannt ist, kann bereits bei der Erzeugung des Feldes dessen Größe so festgelegt werden, daß eine dynamische Vergrößerung des Feldes während der Berechnung nicht notwendig ist. Um die Position des ersten und des letzten Tasks innerhalb einer Warteschlange zu markieren, werden zwei Indexzeiger, `head` und `tail`, verwendet. Zusätzlich wird jeder Warteschlange eine Variable `size` zugeordnet, die zur Abfrage der Größe der Schlange verwendet wird. Da die Nummern der Tasks in der Warteschlange gespeichert werden, können die Tasks in beliebiger Reihenfolge in die Warteschlange eingefügt werden. Weiterhin erlaubt diese Datenstruktur die Entnahme von Tasks an beiden Enden der Warteschlange. Eingefügt werden Tasks immer an der durch `tail` markierten Position. Das Einfügen und die Entnahme von Tasks werden durch Mutex-Variablen (`pthread_spinlock_t`, falls verfügbar; sonst: `pthread_mutex_t`) geschützt.

Die auf Basis dieser Datenstruktur realisierte Lastbalancierungsstrategie ist in Abb. 3.17 dargestellt. Vor Beginn der Berechnung einer Stufe initialisieren die beteiligten Threads ihre lokale Warteschlange, indem sie zunächst `head` und `tail` auf 0 setzen und danach die ihnen zugeordneten Tasks nacheinander an das Ende der Schlange anhängen. Dies geschieht in aufsteigender Reihenfolge der Tasknummern, um ein gutes Lokalitätsverhalten zu erreichen, das mit dem der statischen Arbeitsverteilung vergleichbar ist. Die Variable `size` besitzt währenddessen den Wert 0 und wird zunächst nicht verändert. Anschließend wird die in jeder Stufe erforderliche Barrier-Operation ausgeführt, die hier gleichzeitig genutzt wird, um die Initialisierung der Warteschlange von der Berechnungsphase zu trennen. Erst nach Ausführung der Barrier-Operation wird `size` auf die tatsächliche Größe der Warteschlange gesetzt, wodurch die Größe für andere Threads sichtbar wird. Durch die Geheimhaltung der Größe ist es möglich, eine zweite Barrier-Operation einzusparen, die sonst in jeder Stufe ausgeführt werden müßte, um das Ende der Berechnungsphase von der Initialisierung der Warteschlange für die nächste Stufe zu trennen.

```
1:  for (l := 1; l ≤ s; l++)
2:  {
3:      REWIND(queue[me]);
4:      for (j := first_work_unit[me]; j ≤ last_work_unit[me]; j++)
5:          HIDDEN_APPEND(queue[me], j);
6:      barrier();
7:      REVEAL_SIZE(queue[me]);

8:      loop
9:      {
10:         LOCK(queue[me]);
11:         if (EMPTY(queue[me]))
12:         {
13:             UNLOCK(queue[me]);

14:             loop
15:             {
16:                 sum := Σ_{k=1}^{#threads} SIZE(queue[k]);
17:                 index := argmax_{1≤k≤#threads} SIZE(queue[k]);
18:                 if (sum = 0) goto Stage_Complete;
19:                 if (TRYLOCK(queue[index]))
20:                 {
21:                     if (SIZE(queue[index]) > 0) break;
22:                     UNLOCK(queue[index]);
23:                 }
24:             }

25:             work_units_to_steal := STEAL_HEURISTICS(SIZE(queue[index]), sum);

26:             PREPARE_STEALING(queue[index], work_units_to_steal);
27:             work_unit := STEAL(queue[index]);
28:             REWIND(queue[me]);
29:             for (k := 1; k ≤ work_units_to_steal; k++)
30:                 HIDDEN_APPEND(queue[me], STEAL(queue[index]));
31:             FINISH_STEALING(queue[index], work_units_to_steal);

32:             UNLOCK(queue[index]);
33:             REVEAL_SIZE(queue[me]);
34:         }
35:         else
36:         {
37:             work_unit := FETCH(queue[me]);
38:             UNLOCK(queue[me]);
39:         }

40:         PROCESS_WORK_UNIT(l, work_unit);
41:     }

42:     label Stage_Complete;
43: }
```

Abb. 3.17: Lastbalancierungsstrategie „Task Queue“.

Während der Berechnungsphase entnimmt jeder Thread nacheinander die in seiner lokalen Warteschlange gespeicherten Tasks von deren Anfang und verarbeitet sie. Bezogen auf die Reihenfolge des Einfügens während der Initialisierung werden die Tasks also nach dem FIFO-Prinzip (englisch: *first in first out*) ausgeführt.

Findet ein Thread keine Tasks in seiner lokalen Warteschlange mehr vor, versucht er, Tasks aus einer Warteschlange eines anderen Threads zu stehlen. Dazu liest er die `size`-Variablen aller Warteschlangen und berechnet die Gesamtzahl der noch vorhandenen Tasks und bestimmt den Index der Warteschlange mit der größten Anzahl gespeicherter Tasks. Danach wird versucht, aus dieser Warteschlange mehrere Tasks zu stehlen, deren Anzahl durch eine Heuristik festgelegt wird. Die für die nachfolgend präsentierten Laufzeitexperimente verwendete Heuristik legt diese Anzahl auf die Hälfte der mittleren Größe aller Warteschlangen fest. Da die anderen Threads in der Zwischenzeit jedoch die Verarbeitung von Tasks fortgesetzt haben, muß nach dem Sperren der Quellwarteschlange überprüft werden, ob sich deren Größe verändert hat. Hat sich die Größe der Quellwarteschlange inzwischen verringert, so wird maximal die Hälfte der in der Quellwarteschlange gespeicherten Tasks von deren Ende entnommen und in die leere Zielwarteschlange eingefügt. Die `size`-Variable der Quellwarteschlange wird bereits vor Beginn des Tasktransfers auf die sich nach Abschluß des Tasktransfers ergebende Größe gesetzt, damit die Wahrscheinlichkeit geringer ist, daß andere Threads ebenfalls diese Warteschlange auswählen, um aus ihr Tasks zu stehlen. Analog wird die `size`-Variable der Zielwarteschlange erst nach Abschluß des Tasktransfers auf die neue Größe der Zielwarteschlange gesetzt, um zu verhindern, daß andere Threads bereits während des Tasktransfers versuchen, Tasks daraus zu stehlen. Schlägt das Sperren einer Warteschlange, aus der Tasks gestohlen werden sollen, fehl, wird die Gesamtzahl der Tasks und der Index der größten Warteschlange neu bestimmt und ein neuer Versuch gestartet.

Strategie „Interval Queue“. Zwar bietet die für die Strategie „Task Queue“ realisierte Datenstruktur prinzipiell die Möglichkeit, Tasks in beliebiger Reihenfolge einzufügen, um ein gutes Lokalitätsverhalten zu erhalten, wurde von dieser Möglichkeit jedoch kein Gebrauch gemacht. Stattdessen wurden die Tasks immer geordnet nach aufsteigender Nummer eingefügt. Bei näherer Betrachtung wird deutlich, daß die Warteschlangen zu jedem Zeitpunkt ein zusammenhängendes Intervall von Tasknummern enthalten und daß durch die Entnahme von Tasks lediglich das Intervall verkleinert wird. Dieses Verhalten läßt sich auch mittels einer einfacheren Datenstruktur, deren Manipulation einen deutlich geringeren Overhead erfordert, realisieren. Diese Datenstruktur besteht lediglich aus drei Ganzzahl-Variablen, `first`, `last` und `size`, die die Nummer des ersten und des letzten Tasks sowie die Größe der Warteschlange repräsentieren. Zur Realisierung des wechselseitigen Ausschlusses beim Zugriff auf die Warteschlange werden wie für die Strategie „Task Queue“ ebenfalls Mutex-Variablen aus der POSIX-Thread-Bibliothek verwendet.

Die resultierende Implementierung zeigt Abb. 3.18. Sie arbeitet ähnlich wie die Implementierung der Strategie „Task Queue“, jedoch ergeben sich einige Vereinfachungen. So genügt es zur Entnahme eines Tasks durch den zur Warteschlange gehörigen Thread, die Variable `first` um eins zu erhöhen und die Variable `size` um eins zu verringern. Das Stehlen mehrerer Tasks vom Ende einer Warteschlange geschieht durch Subtraktion der Anzahl der zu stehlenden Tasks von der Variablen `last` und die entsprechende Anpassung der Größe der Warteschlange. Dadurch ist das Stehlen einer beliebigen Anzahl von Tasks in konstanter Zeit möglich, während für die zur Realisierung der Strategie „Task Queue“ verwendete Datenstruktur die Kosten für das Stehlen mehrerer Tasks linear von der Anzahl der Tasks abhängig sind.

Durchführung der Laufzeitexperimente

Zur Untersuchung des Laufzeitverhaltens der Lastbalancierungsimplementierungen wurden Laufzeitexperimente auf vier verschiedenen parallelen Rechnersystemen mit gemeinsamem Adreßraum durchgeführt:

1. auf einem SMP-System vom Typ HP Integrity rx5670, das mit 4 Itanium-2-Prozessoren mit einer Taktfrequenz von mit 1,5 GHz ausgestattet ist,
2. auf einem nach Kundenwünschen konfigurierten Vier-Wege-SMP-System, ausgestattet mit 2 Dual-Core-Opteron-Prozessoren mit einer Taktfrequenz von 2,0 GHz,
3. auf einem Knoten des Supercomputers JUMP, d. h. einem IBM-p690-SMP mit 32 Power4+-Kernen mit einer Taktfrequenz von 1,7 GHz sowie

```
1:  for (l := 1; l ≤ s; l++)
2:  {
3:      REWIND(queue[me], first_work_unit[me], last_work_unit[me]);
4:      barrier();
5:      REVEAL_SIZE(queue[me]);

6:      loop
7:      {
8:          LOCK(queue[me]);
9:          if (EMPTY(queue[me]))
10:         {
11:             UNLOCK(queue[me]);

12:             loop
13:             {
14:                 sum := ∑_{k=1}^{#threads} SIZE(queue[k]);
15:                 index := argmax_{1≤k≤#threads} SIZE(queue[k]);
16:                 if (sum = 0) goto Stage_Complete;
17:                 if (TRYLOCK(queue[index]))
18:                 {
19:                     if (SIZE(queue[index]) > 0) break;
20:                     UNLOCK(queue[index]);
21:                 }
22:             }

23:             work_units_to_steal := STEAL_HEURISTICS(SIZE(queue[index]), sum);

24:             work_unit := STEAL(queue[index], queue[me], work_units_to_steal);

25:             UNLOCK(queue[index]);
26:         }
27:         else
28:         {
29:             work_unit := FETCH(queue[me]);
30:             UNLOCK(queue[me]);
31:         }

32:         PROCESS_WORK_UNIT(l, work_unit);
33:     }

34:     label Stage_Complete;
35: }
```

Abb. 3.18: Lastbalancierungsstrategie „Interval Queue“.

4. auf einem nach Kundenwünschen konfigurierten Vier-Wege-SMP, ausgestattet mit 4 Xeon-MP-Prozessoren mit einer Taktfrequenz von 2,2 GHz, die HyperThreading – eine Variante des simultanen Multithreadings – unterstützen.

Eine detaillierte Beschreibung der betrachteten Rechnersysteme wird in Anhang A gegeben.

Gegenüber der statischen Arbeitsverteilung haben die Lastbalancierungsimplementierung den Nachteil, daß zusätzlicher Programmcode ausgeführt werden muß, der dazu dient, die Lastverteilung zu überprüfen und gegebenenfalls auszugleichen. Sie besitzen aber gleichzeitig den Vorteil, daß die Arbeitsverteilung dynamisch angepaßt und somit ein Leerlaufen von Prozessoren weitestgehend verhindert werden kann. Entscheidend für den Erfolg einer Lastbalancierungsstrategie ist daher, ob bei einer parallelen Ausführung der Laufzeitgewinn durch das Vermeiden des Leerlaufens von Prozessoren den Laufzeitverlust durch die Ausführung des zusätzlichen Programmcodes übertrifft. Zur Bewertung der Lastbalancierungsimplementierungen betrachten wir daher vorrangig den für verschiedene Prozessorzahlen erreichten Speedup, um Skalierbarkeit und Laufzeitverhalten vergleichen zu können. Wir untersuchen aber auch den durch die Ausführung des zusätzlichen Programmcodes entstehenden, sogenannten Instruktionsoverhead bzw. sequentiellen Overhead, indem wir die Implementierungen auf nur einem Prozessor ausführen und die Laufzeitdifferenz zur zugrundeliegenden sequentiellen Implementierung analysieren.

Zur Durchführung dieser Untersuchungen wurden die Lastbalancierungsimplementierungen auf den genannten Maschinen auf verschiedene Testprobleme unter Verwendung des Verfahrens DOPRI5(4) angewandt. Um die für die Durchführung der Laufzeitexperimente erforderliche Zeitdauer weitestgehend zu verkürzen, wurde die Anzahl der ausgeführten Zeitschritte auf 100 festgelegt und die mittlere Laufzeit pro Zeitschritt als Basis für den Vergleich der Implementierungen verwendet. Als Referenz für die Speedup-Berechnung wird die entsprechende Laufzeit der sequentiellen Implementierung (D) verwendet.

HP Integrity rx5670

Als erstes betrachten wir ein SMP-System mit vier Itanium-2-Prozessoren. Auf diesem System ist die Lastbalancierung besonders erfolgreich, da der Maschinenbefehlssatz des Itanium 2 eine atomare Fetch-&-Add-Instruktion enthält, wodurch die auf atomarem Fetch & Inc basierenden Implementierungen sehr effizient realisiert werden können. Weiterhin steht auf diesem System der für unseren Anwendungsfall effizientere Typ `pthread_spinlock_t` für die Realisierung der lockbasierten Implementierungen zur Verfügung. Dieses System besitzt eine dreistufige Cache-Hierarchie. Die größte Cachezeilengröße, die auf diesem System für den L2- und den L3-Cache verwendet wird, beträgt 128 Byte.

Sequentieller Overhead. Bei Betrachtung des in Tab. 3.6 angegebenen sequentiellen Overheads fällt zunächst auf, daß für das Testproblem STARS-MIX und z. T. auch für das Testproblem STARS-CON ein negativer Overhead angegeben ist, was bedeutet, daß die entsprechenden parallelen Implementierungen unerwartet schneller ausgeführt werden konnten als die sequentielle Implementierung (D). Dieser Effekt kann auch auf anderen Maschinen, jedoch z. T. für andere Implementierungen und Testprobleme beobachtet werden. Ursache dafür ist ein verändertes Lokalitätsverhalten. Beispielsweise führt die Arbeitsweise der aktuellen Version der Laufzeitbibliothek, die alle in dieser Arbeit betrachteten Programmcodes enthält, dazu, daß sich zwischen einer sequentiellen und einer zugehörigen parallelen Programmversion eine unterschiedliche Ausrichtung der Datenstrukturen im Speicher ergeben kann. Als Folge davon kann es zu einer Veränderung der Anzahl der Cache-Fehlzugriffe kommen. Auch verändert sich im Zuge der Parallelisierung durch das Hinzufügen von Programmcode die Lage bestimmter Programmteile im Speicher. Dies kann aufgrund von Interferenzen der Speicherzugriffe auf Programmcode und Daten ebenfalls zu einer Veränderung der Anzahl der Fehlzugriffe führen. Geringe Laufzeitunterschiede können auch aus statistischen Schwankungen der Laufzeit bei wiederholter Programmausführung resultieren.

Im Vergleich der verschiedenen Testprobleme bestätigt sich, daß der sequentielle Overhead besonders hoch ist, wenn die Funktionsauswertungskosten gering sind. Dies trifft insbesondere auf die Testprobleme BRUSS2D und MEDAKZO zu, für die ein Overhead bis hin zu mehr als 40 % auftrat. Für EMEP sind die mitteleren Funktionsauswertungskosten dagegen recht hoch, so daß hier nur ein Overhead von weniger als 10 % gemessen wurde. Der geringste Overhead trat für das Testproblem STARS mit $N = 1000$ auf, da hier die Funktionsauswertungskosten am höchsten sind. Hier liegt der Overhead unterhalb von 1 %.

Testproblem	Problemgröße	(D)	(IC)	(IP)	(SCIA)	(SCIL)	(SCRA)	(SPIA)	(SPIL)	(SPRA)	(TC)	(TP)
BRUSS2D-MIX	$N = 250$	3,3	36,3	5,7	11,6	38,0	12,0	3,9	5,0	4,1	42,1	6,4
BRUSS2D-MIX	$N = 1000$	4,9	34,7	7,2	12,9	36,1	12,9	5,7	7,0	5,7	42,5	7,9
BRUSS2D-ROW	$N = 250$	3,2	31,4	4,8	10,3	31,8	10,6	4,0	5,2	3,5	35,2	4,9
BRUSS2D-ROW	$N = 1000$	4,7	29,6	6,2	10,9	29,8	10,9	5,0	6,1	4,9	35,0	6,8
EMEP		1,0	7,4	1,9	2,7	7,7	2,6	1,3	1,9	1,4	8,8	2,1
MEDAKZO	$N = 200$	2,1	18,2	3,9	5,8	19,2	5,9	2,9	3,7	2,8	20,4	3,8
MEDAKZO	$N = 800$	3,4	29,4	6,0	9,4	31,2	9,4	4,5	5,9	4,5	32,7	6,1
MEDAKZO	$N = 2400$	4,2	35,8	7,1	11,4	37,9	11,5	5,4	7,0	5,3	39,8	7,4
STARS-CON	$N = 1000$	0,0	0,2	0,1	0,1	0,2	0,1	0,0	−0,1	0,1	0,1	0,1
STARS-MIX	$N = 1000$	−0,4	−0,2	−0,3	−0,2	−0,3	−0,3	−0,4	−0,2	−0,4	−0,2	−0,2

Tab. 3.6: Sequentieller Overhead der Lastbalancierungimplementierungen und der Implementierung (D) auf einem HP-Integrity-rx5670-Server in %.

Vergleicht man den Overhead der verschiedenen Implementierungsvarianten, so besitzt in aller Regel die Referenzimplementierung (D) mit einer statischen Arbeitsverteilung den geringsten Overhead. Für BRUSS2D beträgt er für die betrachteten Problemgrößen etwa 3–5 %. Den höchsten Overhead generieren die lockbasierten Implementierungen (IC), (TC) und (SCIL), die einzelne Schleifeniterationen als Arbeitseinheiten verwenden. Für BRUSS2D beträgt er etwa zwischen 29 % und 43 %. Die auf atomarem Fetch & Inc basierenden Implementierungen (SCIA) und (SCRA), die ebenfalls einzelne Schleifeniterationen als Arbeitseinheiten verwenden, erzeugen einen deutlich geringeren Overhead, der für BRUSS2D bei ca. 10–13 % liegt. Den geringsten Overhead unter den Lastbalancierungsimplementierungen besitzen die Implementierungen (IP), (TP), (SPIA), (SPRA) und (SPIL), die mehrere Schleifeniterationen zu auf Cachezeilen ausgerichteten Arbeitseinheiten zusammenfassen. Ihr Overhead liegt für BRUSS2D etwa zwischen 4 % und 8 %. Die auf atomarem Fetch & Inc basierenden Implementierungen (SPIA) und (SPRA) sind dabei geringfügig schneller als die lockbasierten Varianten (IP), (TP) und (SPIL).

BRUSS2D. Abbildung 3.19 zeigt die auf dieser Maschine für BRUSS2D-ROW und BRUSS2D-MIX gemessenen Speedup-Werte für die beiden Problemgrößen $N = 250$ und $N = 1000$. Beim Vergleich der in diesen vier Experimenten erreichten Speedup-Werte wird deutlich, daß auf diesem System die Skalierbarkeit eng mit dem Lokalitätsverhalten verknüpft ist. Denn aus diesem Grund werden für $N = 1000$ nur maximale Speedups von 3,32 für BRUSS2D-MIX und 3,36 für BRUSS2D-ROW erreicht, da die gesamte Arbeitsmenge eines Zeitschrittes ca. 153 MB umfaßt und auch bei Aufteilung auf vier Prozessoren nicht in den 6 MB großen L3-Cache paßt. Für $N = 250$ beträgt dagegen die Arbeitsmenge eines Zeitschrittes lediglich rund 9,5 MB. Deshalb kann bereits bei Verwendung von zwei Prozessoren durch die Aufteilung der zu verarbeitenden ODE-Komponenten der größte Teil der Speicherzugriffe aus dem L3-Cache beantwortet werden. Implementierung (D) kann daher für BRUSS2D-MIX bereits mit zwei Prozessoren einen superlinearen Speedup von 2,18 erreichen; für 4 Prozessoren beträgt er 4,33. Für BRUSS2D-ROW wird jedoch nur ein maximaler Speedup von 3,97 erreicht, da die größere Zugriffsdistanz von N^2 (gegenüber $2N$ für BRUSS2D-MIX) zu einer schlechteren Ausnutzung der im Cache gespeicherten Daten führt.

Im Vergleich der Implementierungen lassen sich für die untersuchten Problemgrößen des Testproblems BRUSS2D folgende Aussagen treffen:

- Die Lastbalancierung ist für BRUSS2D-ROW erfolgreicher als für BRUSS2D-MIX, da sich aufgrund der implementierten Fallunterscheidungsstrategien zur Identifikation der Randkomponenten die Funktionsauswertungszeiten von BRUSS2D-MIX für die einzelnen Komponenten weniger stark unterscheiden als für BRUSS2D-ROW. Während die statische Arbeitsverteilung für BRUSS2D-MIX die besten Speedups liefert, können deshalb für BRUSS2D-ROW durch Verwendung der Lastbalancierungsimplementierungen (IP), (SPIA) und (SPRA) höhere Speedups als bei Verwendung einer statischen Arbeitsverteilung erreicht werden.

- Es lohnt sich, eine größere, an der Cachezeilengröße orientierte Taskgranularität zu wählen und mehrere Schleifeniterationen zu einer Arbeitseinheit zusammenzufassen, denn die Implementierungen (IP), (TP), (SPIL), (SPIA) und (SPRA) erreichen bessere Laufzeiten als ihre Gegenstücke (IC), (TC), (SCIL), (SCIA) und (SCRA).

- Der geringere Overhead der Strategie „Interval Queue“ gegenüber der Strategie „Task Queue“ ermöglicht bessere Speedups, insbesondere wenn die Taskgranularität sehr fein ist. So erreicht (IC) deutlich bessere Speedups als (TC), aber auch (IP) erreicht bessere Speedups als (TP). Besonders deutlich ist der Laufzeitunterschied zwischen (IC) und (TC) für BRUSS2D-ROW, da hier im Ausgangszustand ein stärkeres Lastungleichgewicht vorliegt als für BRUSS2D-MIX und deshalb häufiger Tasks gestohlen werden müssen.

- Unter den Implementierungen, die die Strategie „Simple“ umsetzen, sind diejenigen am erfolgreichsten, welche die atomare Maschineninstruktion Fetch & Add zur Realisierung von Fetch & Inc nutzen. Die lockbasierten Implementierungen (SPIL) und (SCIL) sind deshalb deutlich langsamer als ihre Gegenstücke (SPIA) und (SCIA). Insbesondere für (SCIL) wirkt sich die geringe Taskgranularität negativ aus, da die Threads häufiger gleichzeitig versuchen, einen Lock zu sperren.

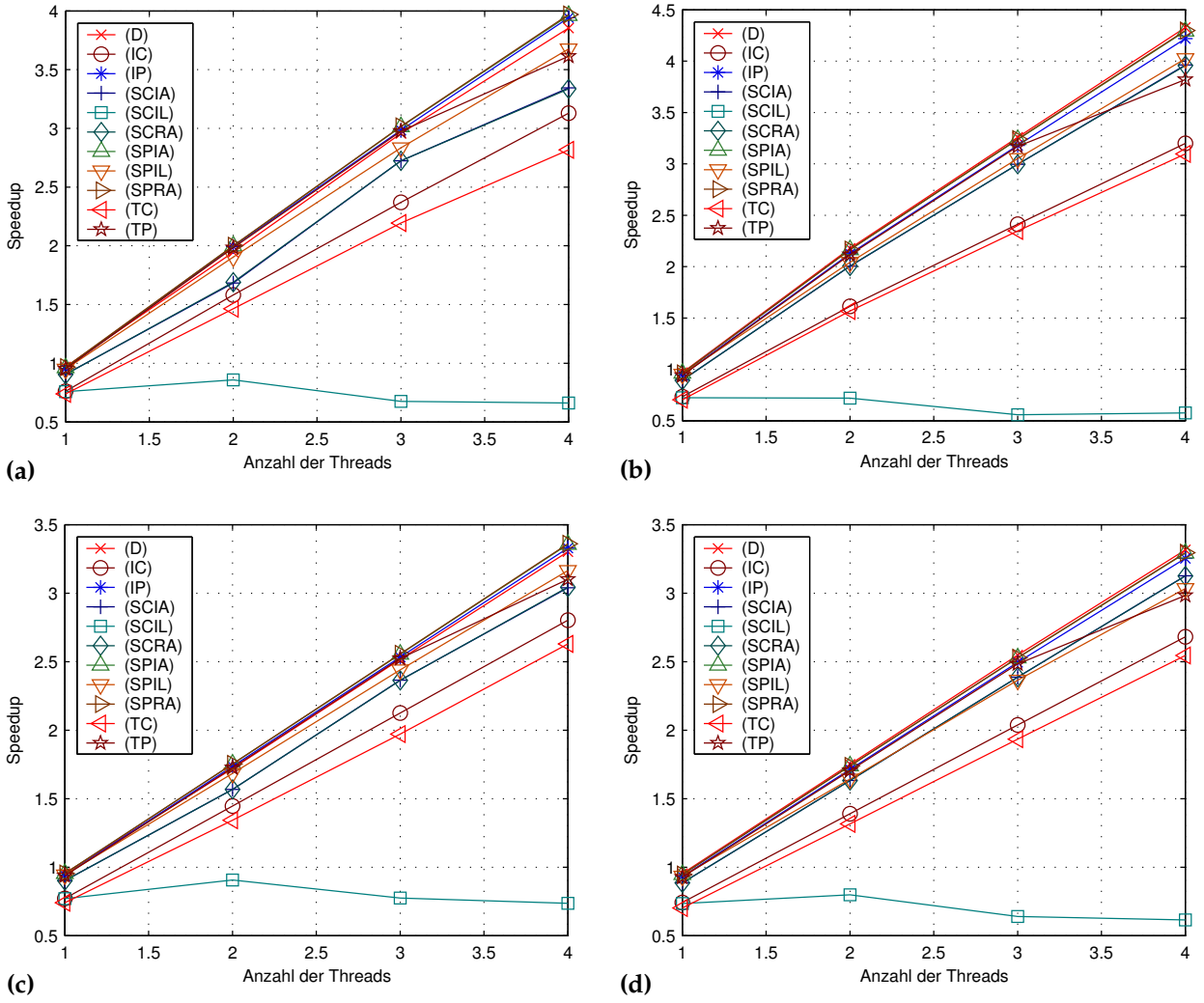


Abb. 3.19: Speedups der Lastbalancierungsimplementierungen im Vergleich zu Implementierung (D) gemessen auf einem HP-Integrity-rx5670-Server. **(a)** BRUSS2D-ROW, $N = 250$, **(b)** BRUSS2D-MIX, $N = 250$, **(c)** BRUSS2D-ROW, $N = 1000$, **(d)** BRUSS2D-MIX, $N = 1000$.

- Die Laufzeitunterschiede zwischen den Strategien „Increment“ und „Random“ zum Wechsel der Thread-ID im Fall der Lastbalancierungsstrategie „Simple“ sind nur sehr gering, so daß keine der beiden Strategien eindeutige Vorteile bietet.

- Da die Implementierungen der Strategien „Task Queue“ und „Interval Queue“ zur Realisierung des wechselseitigen Ausschlusses bezüglich der Warteschlangenzugriffe Locks benutzen, besitzen sie einen höheren sequentiellen Overhead als die Implementierungen der Strategie „Simple“ auf Basis der atomaren Fetch-&-Add-Operation. Dies macht sich insbesondere bei einer feinen Taskgranularität negativ bemerkbar. Infolgedessen erreichen die Implementierungen (SCIA) und (SCRA) deutlich höhere Speedups als (IC), aber auch (SPIA) und (SPRA) sind meßbar schneller als (IP).

EMEP. Aufgrund der kleinen Systemdimension und der hohen Irregularität kann für EMEP keine gute Skalierbarkeit erreicht werden, wie die Darstellung der gemessenen Speedup-Werte in Abb. 3.20 (a) zeigt. Zwar sind die mittleren Funktionsauswertungskosten mit 1650 Takten (gemessen auf einem Pentium-4-System, siehe Abschnitt 3.6.1) im Vergleich zu BRUSS2D und MEDAKZO mit rund 200 Takten recht hoch, absolut gesehen sind sie jedoch bei weitem nicht hoch genug, um für eine effiziente Lastbalancierung einzel-

ne Schleifeniterationen als Arbeitseinheiten verwenden zu können. Die Implementierungen (SCIL), (SCIA), (SCRA), (IC) und (TC) liefern deshalb deutlich schlechtere Speedups als die statische Arbeitsverteilung. Denn während mit einer statischen Arbeitsverteilung zumindest ein maximaler Speedup von 1,14 erzielt werden kann, erreicht die beste Implementierung aus der genannten Gruppe, (IC), nur einen maximalen Speedup von 1,06. Bessere Speedups erreichen die Implementierungen, deren Arbeitseinheiten an der Cachezeilengröße orientiert sind. (IP) und (TP) erreichen beide einen Speedup von 1,12, können also nicht ganz mit der statischen Arbeitsverteilung mithalten. Doch während die statische Arbeitsverteilung ihren maximalen Speedup von 1,14 bei Verwendung von nur zwei Prozessoren erreicht, steigert sich der Speedup von (SPIL), (SPIA) und (SPRA) bis auf 1,16 bis 1,17 bei Verwendung von drei Prozessoren.

Für dieses Testproblem kann also durch den Einsatz von Lastbalancierungstechniken ein besserer Speedup als mit einer statischen Arbeitsverteilung erzielt werden. Aber auch in diesem Fall sind nur geringe Speedups möglich, denn die geringen Funktionsauswertungskosten erfordern eine Vergrößerung der Taskgranularität durch das Zusammenfassen mehrerer Iterationen der Schleifen über die Systemdimension. Da die Systemdimension lediglich 66 beträgt, kann allerdings bereits das Zusammenfassen weniger Komponenten zu einer signifikant unausgewogenen Verteilung des Berechnungsaufwandes führen. Der insgesamt verhältnismäßig geringe Berechnungsaufwand des Berechnungskernels führt außerdem dazu, daß der Laufzeitanteil der Schrittweitenkontrolle an Bedeutung gewinnt.

MEDAKZO. Für MEDAKZO wurden Experimente mit drei verschiedenen Problemgrößen durchgeführt, deren Ergebnisse in Abb. 3.20 (b)–(d) dargestellt sind. Für die betrachteten Problemgrößen verbessert sich die Skalierbarkeit mit wachsender Problemgröße: Für $N = 200$ wird ein maximaler Speedup von 1,41, für $N = 800$ ein maximaler Speedup von 2,21 und für $N = 2400$ ein maximaler Speedup von 3,06 erreicht. Auch der maximale Speedup der statischen Arbeitsverteilung verbessert sich von 1,24 für $N = 200$ auf 1,89 für $N = 800$ und auf 2,74 für $N = 2400$. In allen drei Fällen kann durch Lastbalancierung eine Verbesserung der Skalierbarkeit erreicht werden. Für $N = 200$ liefern sogar nahezu alle Lastbalancierungsimplementierungen einen höheren Speedup als die statische Arbeitsverteilung. Die einzige Ausnahme bildet (SCIL). Der höchste Speedup in diesem Experiment wurde für (IC) gemessen. Für die beiden höheren Problemgrößen können nur die Lastbalancierungsimplementierungen, deren Arbeitseinheiten sich an der Cachezeilengröße orientieren, einen besseren Speedup als die statische Arbeitsverteilung erzielen. Für $N = 800$ wird der höchste Speedup von (SPRA) erreicht; für $N = 2400$ wird er von (IP) erzielt.

Wie für BRUSS2D liefert die Strategie „Interval Queue“ aufgrund des geringeren Overheads bessere Resultate als die Strategie „Task Queue“. Auch die lockbasierten Implementierungen der Strategie „Simple“, (SCIL) und (SPIL), sind langsamer als ihre Gegenstücke auf Basis der atomaren Fetch-&-Add-Instruktion, (SCIA) und (SPIA). Anders als für BRUSS2D kann jedoch (IC) in allen drei Experimenten höhere Speedups erzielen als (SCIA) und (SCRA). Außerdem können geringe Vorteile des randomisierten Wechsels der Thread-ID im Fall von (SCRA) und (SPRA) gegenüber dem inkrementellen Wechsel der Thread-ID im Fall von (SCIA) und (SPIA) festgestellt werden.

STARS. Das Testproblem STARS hat im Gegensatz zu den übrigen betrachteten Beispielen die Eigenschaft, daß die Funktionsauswertungskosten der ODE-Komponenten linear mit der Systemgröße steigen. Das bietet die Möglichkeit, durch Wahl einer geeigneten Problemgröße, wie beispielsweise $N = 1000$, ein grobgranulares Problem zu erzeugen, das im Fall von STARS-CON ein starkes Lastungleichgewicht hervorruft oder im Fall von STARS-MIX sehr gleichmäßig balanciert ist. Die Abbildungen 3.20 (e) und 3.20 (f) zeigen die gemessenen Speedup-Werte für diese beiden Fälle.

Aufgrund der groben Taskgranularität erreichen alle Lastbalancierungsimplementierungen sowohl für STARS-CON als auch für STARS-MIX nahezu perfekte Speedups. Bei Verwendung von p Prozessoren liegen die Speedups für STARS-CON geringfügig unterhalb von p, für STARS-MIX liegen sie dagegen geringfügig oberhalb von p. Diese superlinearen Speedups für STARS-MIX entstehen durch Cache-Effekte. Die Arbeitsmenge eines Zeitschrittes für $N = 1000$ und $p = 1$ beträgt rund 469 KB und kann nicht vollständig im L2-Cache zwischengespeichert werden. Für $p \geq 2$ ist dies jedoch möglich und trägt aufgrund der geringeren Anzahl von Cache-Fehlzugriffen zu einer mehr als linearen Beschleunigung für STARS-MIX bei. Da im Fall von STARS-CON ein höheres Lastungleichgewicht vorliegt, müssen häufiger Tasks migriert werden, um ein Leerlaufen von Prozessoren zu verhindern. Dies führt jedoch zu einem höheren Overhead und

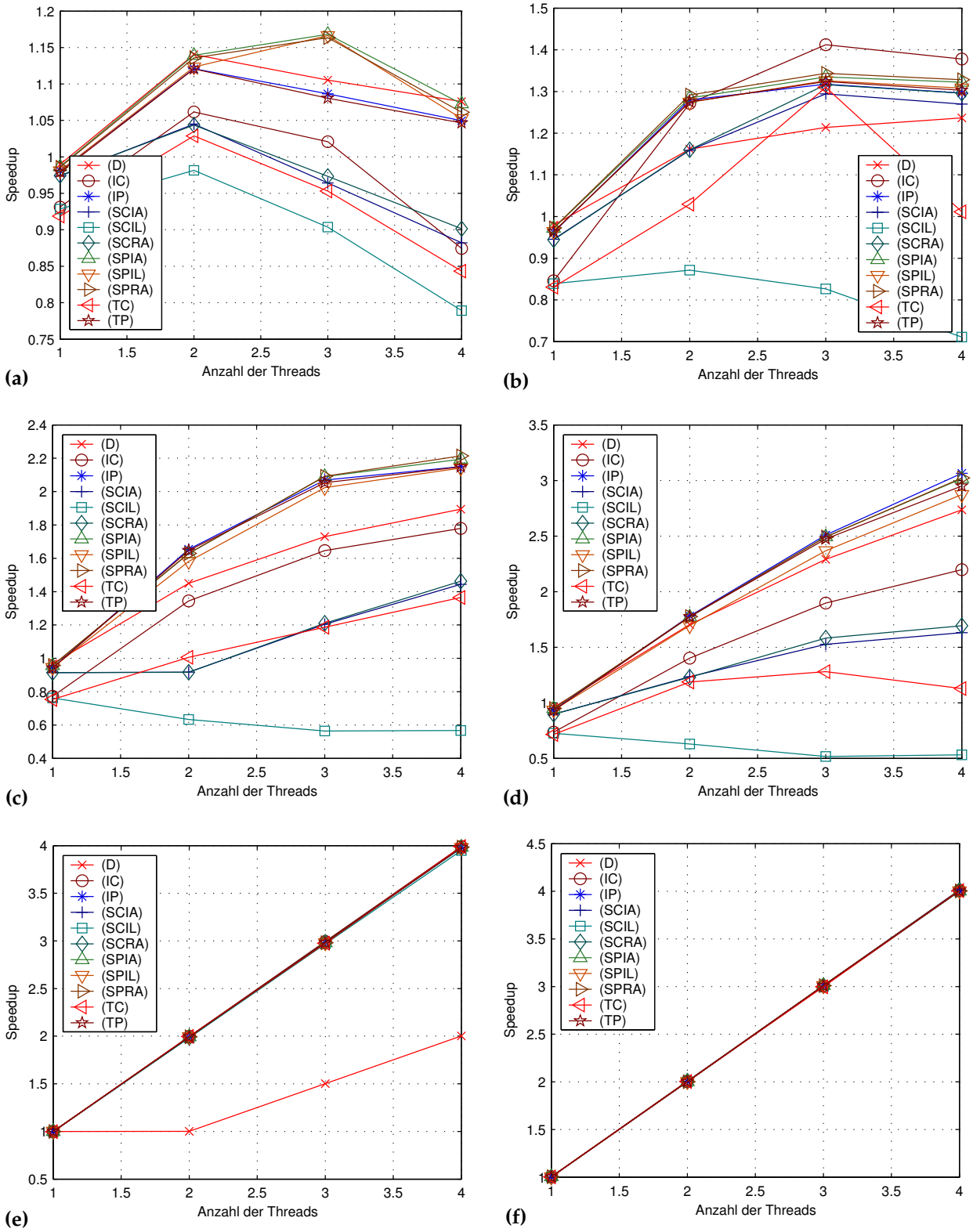


Abb. 3.20: Speedups der Lastbalancierungsimplementierungen im Vergleich zu Implementierung (D) gemessen auf einem HP-Integrity-rx5670-Server. **(a)** EMEP, **(b)** MEDAKZO, $N = 200$, **(c)** MEDAKZO, $N = 800$, **(d)** MEDAKZO, $N = 2400$, **(e)** STARS-CON, $N = 1000$, **(f)** STARS-MIX, $N = 1000$.

schließlich zu geringeren Speedup-Werten. Hinzu kommt, daß für STARS-MIX bereits im sequentiellen Fall ein geringerer Overhead als für STARS-CON gemessen wurde.

Da im Fall von STARS-MIX bereits die Anfangsverteilung eine sehr ausgewogenen Lastbalancierung realisiert, können die Lastbalancierungsimplementierungen die statische Arbeitsverteilung nicht übertreffen, die einen maximalen Speedup von 4,013 erreicht. (SCIA) und (SCRA) sind mit einem Speedup von 4,012 jedoch nur unwesentlich langsamer. Die Anordnung der Komponenten im Fall von STARS-CON führt dazu, daß bei einer blockweisen Anfangsverteilung für große Problemgrößen nahezu der gesamte Berechnungsaufwand mit der zweiten Hälfte der Komponenten, d. h. für $N = 1000$ mit den Komponenten mit Index 3001 bis 6000, verbunden ist. Infolgedessen kann eine statische Arbeitsverteilung für große Problemgrößen bei Verwendung von p Prozessoren nur einen Speedup von etwa $p/2$ erzielen. Deshalb erreichte die statische Arbeitsverteilung in den durchgeführten Experimenten bei Verwendung von vier Prozessoren nur einen Speedup von 2,003. Der Speedup der schnellsten Lastbalancierungsimplementierung, (IC), beträgt dagegen 3,991.

Zwar ist ein Vergleich der Lastbalancierungsimplementierungen aufgrund der eng beieinanderliegenden Laufzeiten schwierig, es können jedoch folgende Tendenzen beobachtet werden:

- Für grobgranulare Probleme erreichen alle Lastbalancierungsimplementierungen gute Speedups, da in allen Fällen der Overhead im Vergleich zur Taskgröße sehr gering ist und daher wenig Einfluß auf die Laufzeit ausübt.

- Da bereits die Funktionsauswertungszeiten für einzelne Komponenten sehr hoch sind, ist es nicht vorteilhaft, mehrere Komponenten zu einer Arbeitseinheit zusammenzufassen. Dies führt lediglich zu einer nicht optimalen Lastbalancierung, ohne den prozentualen Overhead signifikant zu reduzieren.

- STARS-MIX besitzt bereits eine ausgeglichene Anfangsverteilung, so daß kaum Taskmigrationen erforderlich sind. Die besten Speedups erreichen daher die Implementierungen, die unter den Implementierungsvarianten, die einzelne Komponenten als Arbeitseinheiten nutzen, den geringsten sequentiellen Overhead aufweisen. Dies sind die Implementierungen der „Simple“-Strategie (SCIA) und (SCRA).

- Für STARS-CON ist dagegen eine Migration von Tasks für eine effiziente Arbeitsweise zwingend erforderlich. Vorteilhaft sind daher Implementierungsvarianten, die einzelne Komponenten als Tasks verwenden und insgesamt geringe Migrationskosten besitzen. Die besten Speedups werden deshalb von den Implementierungen (IC) und (TC) der Strategien „Interval Queue“ und „Task Queue“ erzielt, da sie die Anzahl der Migrationen reduzieren und dennoch ein weitgehend unabhängiges Arbeiten der Prozessoren ermöglichen.

Ein alternativer Ansatz zum Taskmanagement auf Anwendungsebene besteht darin, pro Prozessor mehrere Threads bzw. insgesamt mehr Threads als Prozessoren zur Verfügung zu stellen und eine Lastbalancierung durch das Scheduling der Threads auf Betriebssystemebene oder durch den Scheduler der Thread-Bibliothek herbeizuführen. Da in diesem Fall der Betriebssystem- bzw. Bibliotheksscheduler zwischen den Threads umschalten, d. h. Kontextwechsel durchführen muß, ist dies ebenfalls mit einem Overhead verbunden. In der Regel ist deshalb das Taskmanagement auf Anwendungsebene erfolgreicher. In bestimmten Fällen, wie z. B. bei der Integration des Testproblems STARS-CON für große Problemgrößen, kann aber auch die Nutzung zusätzlicher Threads zu besseren Speedups führen. Denn startet man auf der betrachteten Vier-Prozessor-Maschine acht Threads bei Verwendung einer statischen Arbeitsverteilung, ohne diese an Prozessoren zu binden, erhält man einen Speedup von 3,993, der den Speedup der besten Lastverteilungsimplementierung von 3,991 geringfügig übertrifft. Dies ist deshalb möglich, weil die vier Threads, die für die Komponenten mit Index 1 bis 3000 zuständig sind, aufgrund der geringen Funktionsauswertungszeiten dieser Komponenten nur für kurze Zeit rechnen und dann die Barrier erreichen. An diesem Punkt werden diese Threads suspendiert, bis die anderen vier Threads die Bearbeitung der Komponenten 3001 bis 6000 abgeschlossen haben, was eine vielfach höhere Rechenzeit erfordert. Der durch Kontextwechsel entstehende Overhead ist daher relativ gering. Die Speedups der Lastbalancierungsimplementierungen ließen sich jedoch verbessern, wenn die Threads an Prozessoren gebunden werden könnten, da in diesem Fall Migrationen von Threads zwischen Prozessoren ausgeschlossen werden und eine bessere Ausnutzung der Cachedaten gegeben ist.

Opteron-SMP

Das zweite Zielsystem, das für die Untersuchung der Lastbalancierungsstrategien verwendet wurde, ist ebenfalls ein Vier-Wege-SMP, das jedoch mit zwei Dual-Core-Prozessoren vom Typ AMD Opteron DP 270 mit einer Taktfrequenz von 2 GHz ausgestattet ist. Die Cache-Hierarchie ist zweistufig. Jeder Prozessorkern besitzt einen eigenen L2-Cache mit einer Größe von 1 MB. Alle Caches verwenden eine Zeilengröße von 64 Byte.

Auch auf dieser Maschine kann `pthread_spinlock_t` für die Realisierung der lockbasierten Implementierungen verwendet werden. Eine atomare Fetch-&-Add-Instruktion steht jedoch nicht zur Verfügung, so daß für die Strategie „Simple“ Fetch & Inc emuliert werden muß, indem in einer Schleife mehrfach versucht wird, die entsprechende Speicherzelle auszulesen, zu inkrementieren und mittels Compare & Swap zurückzuschreiben. Im Gegensatz zum Itanium-2-System bestand auf diesem System die Möglichkeit, Threads an Prozessoren zu binden. Diese Möglichkeit wurde genutzt, da sich so höhere Speedups erzielen ließen, weil ausgeschlossen wird, daß durch eine Migration eines Threads auf einen anderen Prozessor die Daten im Cache des bisher genutzten Prozessors für eine Wiederverwendung verloren gehen.

Sequentieller Overhead. Die Emulation von Fetch & Inc durch eine Schleife führt dazu, daß sich der sequentielle Overhead (Tab. 3.7) der Implementierungen (SCIA), (SCRA), (SPIA) und (SPRA) vergrößert und ungefähr gleichauf mit dem der lockbasierten Varianten mit gleichen Arbeitseinheiten, (IC), (TC) und (SCIL) sowie (IP), (TP) und (SPIL), liegt. Insgesamt betrachtet sind die auf dem Opteron-SMP beobachteten Overheads höher als auf dem Itanium-2-SMP. Während dort beispielsweise für BRUSS2D die Implementierungen (IC), (TC) und (SCIL) den höchsten Overhead generierten, der zwischen 29 % und 43 % lag, erzeugen diese Implementierungen und auch die Implementierungen (SCIA) und (SCRA) auf dem Opteron-SMP einen Overhead zwischen ungefähr 37 % und 86 %.

BRUSS2D. Aufgrund des höheren sequentiellen Overheads ist es auf dem Opteron-SMP schwieriger, durch den Einsatz von Lastbalancierungsstrategien bessere Speedups als für eine statische Arbeitsverteilung zu erzielen. Für die Implementierungen (SCIA), (SCRA), (SPIA) und (SPRA) kommt hinzu, daß Fetch & Inc durch die Emulation mittels Compare & Swap nun ähnliche Eigenschaften wie ein Lock aufweist und deshalb ein höherer paralleler Overhead entsteht.

Eine Auswahl der auf dem Opteron-SMP für BRUSS2D gemessenen Speedup-Werte ist in Abb. 3.21 dargestellt. Für die dargestellten Problemgrößen $N = 250$ und $N = 1000$ wurden im Vergleich zu den Experimenten auf dem Itanium-2-SMP wesentlich geringere Speedup-Werte gemessen. Dies ist zu einem großen Teil auf eine schlechtere Ausnutzung der Lokalität aufgrund der geringeren Cachegröße zurückzuführen, denn auch für die kleinere Problemgröße von $N = 250$ beträgt die Arbeitsmenge eines Zeitschrittes rund 9,5 MB und kann daher auch bei einer Aufteilung auf vier Prozessoren nicht im 1 MB großen L2-Cache vorgehalten werden. Dies wiegt um so schwerer, da der Opteron eine wesentlich geringere sequentielle Laufzeit erreicht (z. B. 67 ms pro Zeitschritt gegenüber 142 ms pro Zeitschritt für BRUSS2D-MIX, $N = 250$), pro Zeiteinheit also mehr Speicheroperationen ausgeführt werden. Dementsprechend höher sind bei einer parallelen Ausführung die Anforderungen an die Skalierbarkeit der Verbindung zwischen Speicher und Prozessoren.

Wie auf dem Itanium-2-SMP ist (SCIL) auch auf dem Opteron-SMP nicht in der Lage, durch die parallele Ausführung einen Gewinn gegenüber der sequentiellen Implementierung zu erzielen. Alle übrigen Implementierungen besitzen eine bessere Skalierbarkeit. Die Speedup-Kurven flachen jedoch schneller ab und erreichen nicht die gleiche Linearität wie auf dem Itanium-2-SMP. Bei Verwendung von nur einem oder zwei Prozessoren führt die statische Arbeitsverteilung zur geringsten Laufzeit. Um mittels Lastbalancierung die Performance der statischen Arbeitsverteilung erreichen oder übertreffen zu können, ist die Verwendung von drei bzw. vier Prozessoren erforderlich. Da (SPIA) und (SPRA) durch die Emulation von Fetch & Inc ein lockähnliches Verhalten aufweisen, können sie (IP) nicht mehr übertreffen, erreichen aber annähernd gleiche Speedups. Allerdings kann nur (IP) in allen vier dargestellten Experimenten die statische Arbeitsverteilung bei Verwendung von bis zu vier Threads übertreffen. So erreicht (IP) für BRUSS2D-ROW und BRUSS2D-MIX für $N = 250$ einen Speedup von 1,94 bzw. 2,26 und für $N = 1000$ einen Speedup von 2,11 bzw. 2,28, während die statische Arbeitsverteilung unter diesen Voraussetzungen nur Speedups von 1,93 und 2,22 bzw. 2,10 und 2,26 erreicht.

Testproblem	Problemgröße	(D)	(IC)	(IP)	(SCIA)	(SCIL)	(SCRA)	(SPIA)	(SPIL)	(SPRA)	(TC)	(TP)
BRUSS2D-MIX	$N = 250$	−0,8	55,2	6,2	54,7	57,4	53,9	7,4	8,0	7,3	69,0	8,6
BRUSS2D-MIX	$N = 1000$	−1,3	40,6	4,2	40,0	41,9	39,5	6,3	6,4	6,1	55,9	7,5
BRUSS2D-ROW	$N = 250$	1,4	69,6	14,9	74,0	73,1	73,2	16,5	16,6	16,1	85,3	17,3
BRUSS2D-ROW	$N = 1000$	0,4	37,7	7,7	41,5	40,7	41,1	9,0	9,3	9,1	53,1	9,8
EMEP		0,5	3,8	2,5	3,9	5,6	4,3	2,3	3,3	2,7	4,0	2,6
MEDAKZO	$N = 200$	2,2	26,9	7,3	24,1	31,5	22,2	9,2	9,5	10,3	29,4	8,6
MEDAKZO	$N = 800$	−0,7	32,8	7,2	31,4	38,8	29,5	9,3	10,0	11,0	36,8	8,8
MEDAKZO	$N = 2400$	0,5	34,9	7,6	33,4	40,4	31,4	9,8	10,5	11,8	40,2	9,5
STARS-CON	$N = 1000$	0,0	0,1	0,1	0,2	0,2	−0,1	0,1	0,1	0,1	0,1	0,1
STARS-MIX	$N = 1000$	0,0	0,2	0,1	0,3	0,3	0,3	0,1	0,1	0,1	0,2	0,1

Tab. 3.7: Sequentieller Overhead der Lastbalancierungimplementierungen und der Implementierung (D) auf einem Vier-Wege-Opteron-SMP in %.

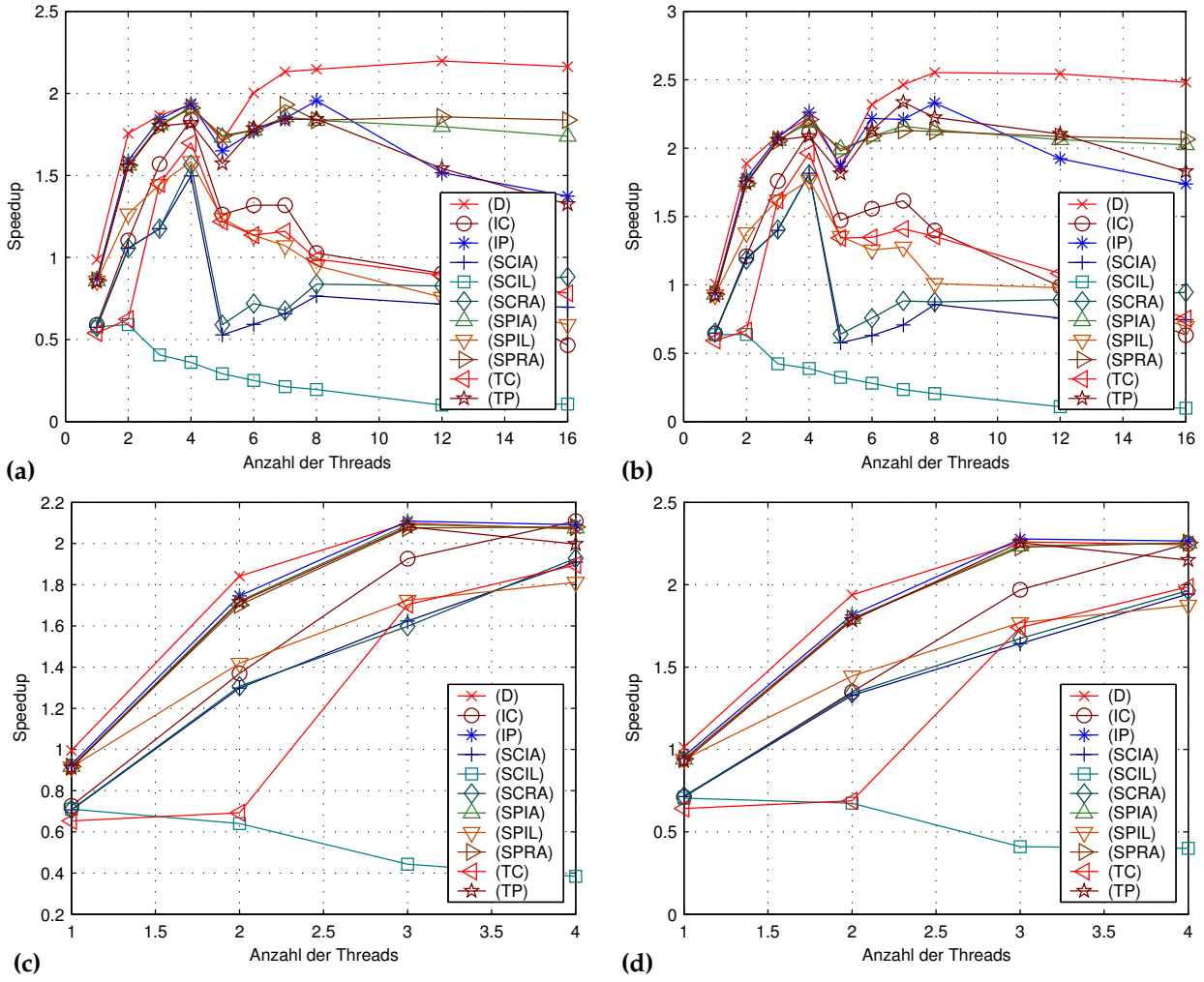


Abb. 3.21: Speedups der Lastbalancierungsimplementierungen im Vergleich zu Implementierung (D) gemessen auf einem Vier-Wege-Opteron-SMP. **(a)** BRUSS2D-ROW, $N = 250$, **(b)** BRUSS2D-MIX, $N = 250$, **(c)** BRUSS2D-ROW, $N = 1000$, **(d)** BRUSS2D-MIX, $N = 1000$.

Steigert man jedoch die Anzahl der Threads über die Prozessoranzahl von vier hinaus, kann die statische Arbeitsverteilung ihre Leistung für bestimmte Problemgrößen, u. a. $N = 250$, verbessern und erreicht dadurch signifikant höhere Speedups als die Lastbalancierungsimplementierungen. Beispielsweise wurde für $N = 250$ für BRUSS2D-ROW ein maximaler Speedup von 2,20 bei Verwendung von 12 Threads und für BRUSS2D-MIX ein maximaler Speedup von 2,55 bei Verwendung von acht Threads gemessen. Diese Verbesserung des Speedups ist allerdings nur unter der Voraussetzung möglich, daß die Threads an Prozessoren gebunden werden, wobei die Datenbereiche der auf einem Prozessor arbeitenden Threads gemeinsam einen zusammenhängenden Datenbereich bilden. Dadurch ist es möglich, daß Threads Daten wiederverwenden können, die durch Speicherzugriffe eines anderen Threads in den Cache geladen wurden, was zu einer Verbesserung des Lokalitätsverhaltens führt. Auch können zusätzliche Threads z. T. eine Verbesserung der Lastbalancierung bewirken. Für andere Problemgrößen kann sich dieser Effekt allerdings umkehren, wenn z. B. ein Thread von einem anderen Thread in den Cache geladene Daten überschreibt, die dieser andere Thread hätte wiederverwenden können. Doch auch wenn keine Verschlechterung des Lokalitätsverhaltens auftritt, sondern die Gesamtzahl der Cache-Fehlzugriffe bei Erhöhung der Threadanzahl etwa konstant bleibt, erhöht sich der Overhead durch zusätzliche Kontextwechsel, was in der Regel zu einer Erhöhung der Laufzeit führt.

Im Vergleich zwischen den Strategien „Task Queue“ und „Interval Queue“ weist die Strategie „Interval Queue“ auch auf dem Opteron-SMP die geringeren Laufzeiten auf. Wiederum ist der Unterschied besonders deutlich bei einer geringen Taskgranularität zu beobachten, da in diesem Fall die Anzahl der zu migrierenden Tasks und folglich auch der Overhead von (TC) gegenüber (IC) höher ist. Für BRUSS2D-ROW und $N = 1000$ kann (IC) sogar den gleichen Speedup wie (IP) von 2,11 erreichen und somit alle anderen Implementierungen, einschließlich der statischen Arbeitsverteilung, übertreffen. In anderen Experimenten, insbesondere für $N = 250$, ist (IC) dagegen langsamer als die statische Arbeitsverteilung. Die Implementierungen mit höherer Taskgranularität, (TP) und (IP), erreichen für kleine Prozessorzahlen annähern gleiche Speedups, wobei (IP) jedoch meist geringfügig schneller als (TP) ist. Bei Verwendung von vier Prozessoren kann sich (IP) jedoch in allen dargestellten Experimenten deutlich absetzen.

Die Strategie „Simple“ ist weniger erfolgreich als auf dem Itanium-2-SMP. Zwar erreicht auch die Emulation von Fetch & Inc mittels Compare & Swap immer noch ein erheblich besseres Skalierbarkeitsverhalten als die entsprechenden lockbasierten Varianten (SCIL) und (SPIL), im Vergleich zu (IP) und (TP) bzw. (IC) und (TC) schneiden (SPIA) und (SPRA) bzw. (SCIA) und (SCRA) jedoch meist schlechter ab. Lediglich in einigen Fällen kann die Strategie „Task Queue“ übertroffen werden, die effizientere Strategie „Interval Queue“ erreicht jedoch in allen Experimenten einen höheren maximalen Speedup.

EMEP. Wie Abb. 3.22 (a) zeigt, erreichen für EMEP alle Implementierungen bessere Speedup-Werte als auf dem Itanium-2-SMP. Da EMEP nur eine Systemgröße von 66 besitzt, beträgt die Größe der Arbeitsmenge eines Zeitschrittes lediglich 5,2 KB. Im wesentlichen führen daher nur die zwanghaften Fehlzugriffe, die beim erstmaligen Zugriff auf ein Datenelement auftreten, zu Hauptspeicherzugriffen. Die Skalierbarkeit wird daher wesentlich durch die Interprozessorkommunikation und weniger durch die Speicheranbindung bestimmt. Der Einsatz von Lastbalancierungstechniken ist für EMEP auf dem Opteron-SMP erfolgreicher als auf dem Itanium-2-SMP. Hier erreichen mit Ausnahme von (SCIL) alle Lastbalancierungsimplementierungen höhere Speedups als die statische Arbeitsverteilung, für die ein maximaler Speedup von 1,37 gemessen wurde. Auch können bis auf (SCIL) alle Implementierungen von der Verwendung von drei Prozessoren profitieren, was auf dem Itanium-2-SMP nur auf (SPIL), (SPIA) und (SPRA) zutrifft. Der höchste Speedup von 1,50 wurde für (SPIA) gemessen. (SCIA), (SCRA), (SPIL) und (SPRA) sind mit Speedups zwischen 1,48 und 1,49 jedoch nur wenig langsamer. Für die drei Implementierungen (IP), (TC) und (TP) wurde ein Speedup von rund 1,45 gemessen. (IC) erreicht einen maximalen Speedup von 1,43. Am schlechtesten skaliert, wie bereits erwähnt, (SCIL) mit einem Speedup von 1,25. Wie auf dem Itanium-2-SMP ist also auch auf dem Opteron-SMP die Strategie „Simple“ am erfolgreichsten.

MEDAKZO. Für MEDAKZO ergeben sich für die drei Problemgrößen $N = 200$, $N = 800$ und $N = 2400$ ähnliche hohe maximale Speedups wie auf dem Itanium-2-SMP (siehe Abb. 3.22 (b)–(d)). Vergleicht man jedoch die Speedups der statischen Arbeitsverteilung, so sind diese höher als auf dem Itanium-2-SMP. Im Unterschied zu diesem können die Lastbalancierungsimplementierungen die statische Arbeitsverteilung daher nicht in gleichem Maße übertreffen. Für $N = 200$ beträgt der maximale Speedup der statischen Arbeitsverteilung 1,28. Er wird bei Verwendung von drei Prozessoren erreicht. Die beste Lastbalancierungsimplementierung für diese Problemgröße ist (SPRA) mit einem Speedup von 1,34, der ebenfalls bei Verwendung von drei Prozessoren erreicht wird. Nur wenig langsamer sind (IP), mit einem Speedup von 1,33, sowie (SPIA) und (TP), die beide einen Speedup von 1,32 erreichen. Alle übrigen Implementierungen liefern für diese Problemgröße geringere Speedups als die statische Arbeitsverteilung. Bei Steigerung der Problemgröße verschlechtert sich die Geschwindigkeit der Lastbalancierungsimplementierungen im Vergleich zur statischen Arbeitsverteilung. So übertrifft für $N = 800$ nur (SPIA) deren maximalen Speedup; für $N = 2400$ gelingt es keiner der Lastbalancierungsimplementierungen, einen höheren Speedup als die statische Arbeitsverteilung zu erzielen. In allen drei Experimenten hat sich eine feine Taskgranularität als ungünstig erwiesen. Denn während (IP), (TP), (SPIA) und (SPRA) ähnlich gute oder gar bessere Speedups erreichen wie die statische Arbeitsverteilung, sind (IC), (TC), (SCIA) und (SCRA) deutlich langsamer als diese. Wie in allen vorherigen Experimenten schneiden die lockbasierten Implementierungen der Strategie „Simple“ schlechter ab als ihre Gegenstücke auf Basis von Fetch & Inc.

STARS. Aufgrund der hohen Funktionsauswertungskosten unterscheidet sich der Speedup-Graph für die Integration der beiden Testprobleme STARS-CON und STARS-MIX auf dem Opteron-SMP (Abb. 3.22 (e)

und 3.22 (f)) visuell kaum von den Speedup-Graphen des Itanium-2-SMP. Auch hier erreichen alle Lastbalancierungsimplementierungen nahezu ideale Speedups zwischen 3,953 und 3,984 für STARS-MIX und zwischen 3,913 und 3,983 für STARS-CON. Damit übertreffen sie in beiden Fällen die statische Arbeitsverteilung, die bei Verwendung von vier Threads für STARS-MIX einen Speedup von 3,939 und für STARS-CON einen Speedup von 1,975 erreicht. Zwar läßt sich auch auf dem Opteron-SMP der Speedup der statischen Arbeitsverteilung für STARS-CON durch die Verwendung zusätzlicher Threads verbessern, durch die Bindung von Threads an Prozessoren erfolgt jedoch keine so gute Lastbalancierung wie auf dem Itanium-2-SMP. Der maximale Speedup der statischen Arbeitsverteilung beträgt daher nur 2,956 und wurde bei Verwendung von 6 Threads erzielt.

Doch auch wenn sich die Speedups der Lastbalancierungsimplementierungen nur geringfügig unterscheiden, lassen sich dennoch Tendenzen erkennen, die einen Vergleich der verschiedenen Lastbalancierungsstrategien ermöglichen. Für STARS-CON liefern (SCIL), (SCIA) und (SCRA) die schlechtesten Speedups von 3,913 bis 3,927. Die übrigen Implementierungen der Strategie „Simple“, deren Arbeitseinheiten sich an der Cachezeilengröße orientieren, (SPIL), (SPIA) und (SPRA), erreichen erkennbar bessere Speedups zwischen 3,965 und 3,973. Die höchsten Speedups erzielen die Implementierungen der Strategien „Interval Queue“ und „Task Queue“, wobei (TP) und (IP) einen Speedup von 3,977 bzw. 3,978 und (TC) und (IC) einen Speedup von 3,982 bzw. 3,983 erreichen. Für STARS-MIX ergibt sich eine etwas anderer Reihenfolge. Zwar ist auch hier (SCIL) mit einem Speedup von 3,593 am langsamsten, (SPIA) und (SPRA) sind jedoch mit einem Speedup von 3,983 bzw. 3,982 geringfügig schneller als (SPIL), (SCIA) und (SCRA) mit Speedups von 3,977, 3,979 bzw. 3,980. Die schnellste Gruppe bilden (TC), (SPRA), (SPIA), (IC), (TP) und (IP) mit Speedups zwischen 3,982 und 3,984, wobei (IP) den höchsten Speedup erzielt. Im Unterschied zu STARS-CON gibt also hier der geringere Overhead von (IP) und (TP) gegenüber (IC) und (TC) den Ausschlag, während für STARS-CON die feinere Granularität von (IC) und (TC) von Vorteil ist. Anders verhält es sich jedoch für die Strategie „Simple“. Hier ist sowohl für STARS-CON als auch für STARS-MIX eine höhere Granularität vorteilhaft, da eine Verringerung der Granularität das Konfliktpotential für Zugriffe auf den Zähler verstärkt, wenn mehrere Threads die gleiche Thread-ID annehmen. Bei Verwendung der Strategien „Task Queue“ und „Interval Queue“ kommen derartige Konflikte weniger häufig vor, da nur während des Stehlens mehrere Threads auf die gleiche Warteschlange zugreifen.

IBM p690

Um die Skalierbarkeit der Lastbalancierungsimplementierungen für eine größere Anzahl von Prozessoren untersuchen zu können, wurden weitere Laufzeitexperimente auf einem Knoten des Supercomputers JUMP, d. h. einem IBM-p690-SMP mit 32 Power4+-Prozessorkernen, durchgeführt. Die Prozessoren arbeiten mit einer Taktfrequenz von 1,7 GHz. Je zwei Prozessorkerne sind auf einem einzelnen Chip integriert. Pro Chip ist ein 1,5 MB großer L2-Cache mit einer Zeilengröße von 128 Byte vorhanden, den sich die beiden Prozessorkerne teilen. Weiterhin befindet sich zwischen jedem Chip und dem ihm zugeordneten Speicher ein 32 MB großer L3-Cache, der so konfiguriert werden kann, daß er zusammen mit anderen L3-Caches einen von mehreren Chips gemeinsam genutzten, größeren L3-Cache mit höherer Bandbreite bildet. Der L3-Cache verwendet Zeilen mit einer Größe von 512 Byte, die in vier 128 Byte große Sektoren unterteilt sind, auf deren Basis die Cache-Kohärenz gewährleistet wird.

Der Typ `pthread_spinlock_t` steht auf dieser Maschine nicht zur Verfügung, weshalb zur Realisierung der Locks innerhalb der Implementierungen (IC), (IP), (TC), (TP), (SCIL) und (SPIL) auf den Typ `pthread_mutex_t` zurückgegriffen wurde. Zur Realisierung von Fetch & Inc in den Implementierungen (SCIA), (SPIA), (SCRA) und (SPRA) wurde die AIX-Kernelfunktion `fetch_and_add()` verwendet, da dies die Erstellung der Implementierungen wesentlich vereinfachte und dennoch einen geringen Overhead versprach. Es besteht jedoch die Möglichkeit, Fetch & Inc mit Hilfe der Maschineninstruktionen Load & Reserve und Store Conditional zu emulieren und dadurch den Overhead weiter zu verringern. Eine Bindung von Threads an Prozessoren wurde nicht vorgenommen.

Für die Implementierungen (IP), (TP), (SPIL), (SPIA) und (SPRA), deren Arbeitseinheiten sich an der Cachezeilengröße orientieren, wurde eine Cachezeilengröße von 128 Byte vorgegeben, da sowohl der L2- als auch der L3-Cache die Cache-Kohärenz auf Basis von 128 Byte großen Blöcken sicherstellt. Nur für BRUSS2D wurden mehrere Experimente mit unterschiedlichen Vorgaben für die Zeilengröße durchgeführt, um deren Einfluß auf Laufzeit und Skalierbarkeit zu untersuchen. Für die übrigen Testprobleme wurde

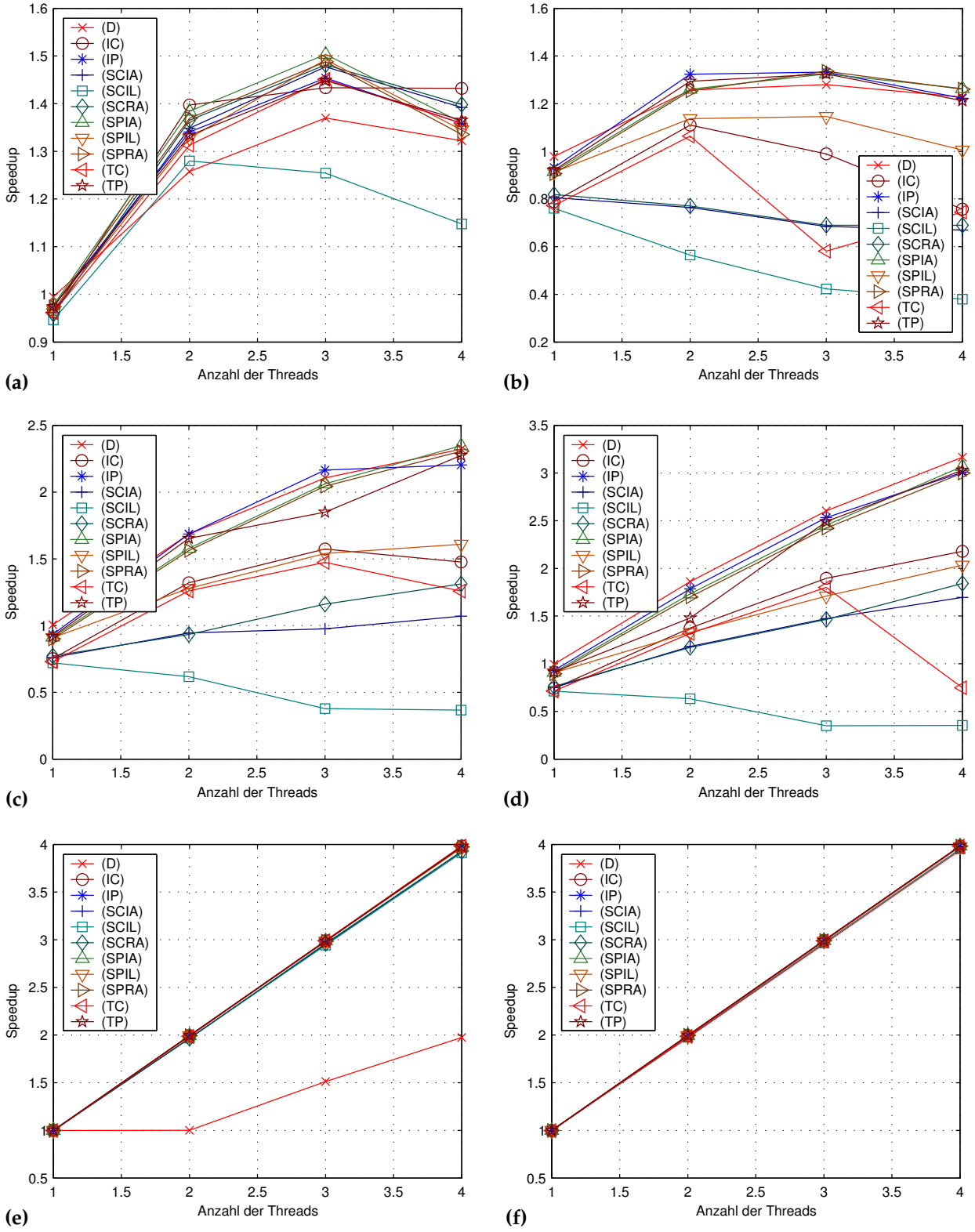


Abb. 3.22: Speedups der Lastbalancierungsimplementierungen im Vergleich zu Implementierung (D) gemessen auf einem Vier-Wege-Opteron-SMP. **(a)** EMEP, **(b)** MEDAKZO, $N = 200$, **(c)** MEDAKZO, $N = 800$, **(d)** MEDAKZO, $N = 2400$, **(e)** STARS-CON, $N = 1000$, **(f)** STARS-MIX, $N = 1000$.

darauf verzichtet, da die Vorgabe einer höheren Zeilengröße mit einer groberen Taskgranularität einhergeht und aufgrund der im Vergleich zu BRUSS2D geringen Systemdimension zu einem nichtbehebbaren Lastungleichgewicht führen würde.

Sequentieller Overhead. Wie Tab. 3.8 zeigt, besitzen die Lastbalancierungsimplementierungen in der Regel einen teilweise um ein Vielfaches höheren sequentiellen Overhead als auf dem Opteron- und dem Itanium-2-SMP. In einigen Fällen kann der sequentielle Overhead aber auch geringer sein als etwa auf dem Opteron-SMP. Besonders problematisch sind die lockbasierten Implementierungen (IC), (IP), (TC), (TP), (SCIL) und (SPIL), für deren Realisierung auf Locks des Typs `pthread_mutex_t` zurückgegriffen wurde. Während z. B. für MEDAKZO auf dem Opteron- und dem Itanium-2-SMP der sequentielle Overhead der Implementierungen (IC), (TC) und (SCIL) im schlechtesten Fall geringfügig mehr als 40 % betrug, beläuft er sich hier auf über 200 %. Gute Speedup-Werte sind daher für die lockbasierten Implementierungen nicht zu erwarten. Doch auch der Overhead der Implementierungen der Strategie „Simple“, die die Kernelfunktion `fetch_and_add()` zur Realisierung von Fetch & Inc nutzen, erhöht sich in vielen Fällen. So erzeugen in den Experimenten mit BRUSS2D alle diese Implementierungen einen höheren sequentiellen Overhead als auf dem Itanium-2-SMP. (SCIA) und (SCRA) sind für BRUSS2D auch im Vergleich zum Opteron-SMP erheblich langsamer. Für EMEP ist der sequentielle Overhead ungefähr gleich hoch wie auf dem Opteron-SMP; für STARS ist er teilweise sogar geringer. Der Overhead für MEDAKZO ist für (SPIA) höher als auf dem Opteron-SMP; für (SCIA), (SCRA), (SPRA) fällt er dagegen geringer aus.

BRUSS2D. Abbildung 3.23 zeigt die für das Testproblem BRUSS2D für verschiedene Vorgaben der Cachezeilengröße gemessenen Speedup-Werte. Die statische Arbeitsverteilung, die unabhängig von der vorgegebenen Zeilengröße arbeitet, liefert eine als gut zu bewertende Skalierbarkeit. Für BRUSS2D-MIX erreicht sie bei Verwendung von 32 Prozessoren einen Speedup von rund 25,4 und für BRUSS2D-ROW sogar einen Speedup von 27,2, was von keiner anderen Implementierung übertroffen wird. Alle Lastbalancierungsimplementierungen, die einzelne Iterationen der Schleifen über die Systemdimension als Arbeitseinheiten verwenden und daher ebenfalls unabhängig von der Vorgabe der Zeilengröße arbeiten, erreichen nur relativ geringe Speedups. (SCIL) ist beispielsweise nicht einmal in der Lage, bei Verwendung mehrerer Prozessoren die sequentielle Laufzeit zu unterbieten. (IC) und (TC) können zwar von einer parallelen Ausführung profitieren, erreichen bei Verwendung von 12 Prozessoren jedoch nur Speedups von rund 2,6 bzw. 2,4 für BRUSS2D-MIX und 2,8 bzw. 2,6 für BRUSS2D-ROW. (SCIA) und (SCRA) arbeiten etwas effizienter, erreichen aber auf 12 Prozessoren ebenfalls nur Speedups von rund 6,0 bzw. 6,2 für BRUSS2D-MIX und 5,7 bzw. 6,1 für BRUSS2D-ROW.

Die Lastbalancierungsimplementierungen (SPIL), (SPIA), (SPRA), (TP) und (IP), die mehrere Schleifeniterationen zu einer Arbeitseinheit zusammenfassen, so daß die zugehörigen Vektorblöcke die vorgegebene Cachezeilengröße besitzen, können zwar eine ähnlich gute Skalierbarkeit erreichen wie die statische Arbeitsverteilung, besitzen aber einen zu großen Overhead, um diese übertreffen zu können. Der sequentielle Overhead und auch die Konfliktwahrscheinlichkeit für den gleichzeitigen Zugriff auf Locks oder für die gleichzeitige Ausführung von `fetch_and_add()` kann zwar durch Erhöhung der Vorgabe der Zeilengröße verringert werden, durch die resultierende Erhöhung der Taskgranularität verschlechtert sich jedoch die Lastbalancierung.

Bei Vorgabe einer Zeilengröße von 128 Byte ergibt sich bereits für kleine Prozessorzahlen ein deutlicher Rückstand der besten Lastbalancierungsimplementierung, (SPRA), auf die statische Arbeitsverteilung. Bei Verwendung von 32 Prozessoren erreicht (SPRA) nur einen Speedup von 23,5 für BRUSS2D-MIX und 25,3 für BRUSS2D-ROW. Die Implementierung (SPIA), die auf allen zuvor betrachteten Rechnersystemen etwa gleichauf mit (SPRA) war, bleibt auf diesem System für große Prozessorzahlen deutlich erkennbar hinter (SPRA) zurück. Für BRUSS2D-MIX erreicht (SPIA) nur einen Speedup von 23,1; für BRUSS2D-ROW sogar nur einen im Verhältnis schlechteren Speedup von 24,0. (IP) und (TP) liefern noch schlechtere Speedups. Für BRUSS2D-MIX erreicht (IP) nur einen Speedup von 21,1; für BRUSS2D-ROW beträgt er 22,6. Die Speedup-Kurve von (TP) zeigt einen Einbruch bei Verwendung von 28 Prozessoren. Der maximale Speedup von 17,2 für BRUSS2D-MIX und 18,5 für BRUSS2D-ROW wird deshalb mit nur 24 Prozessoren erreicht. (SPIL) kann nur bis zu vier Prozessoren für eine Beschleunigung der Ausführung nutzen. Für größere Prozessorzahlen verschlechtert sich der Speedup von (SPIL).

Testproblem	Problemgröße	(D)	(IC)	(IP)	(SCIA)	(SCIL)	(SCRA)	(SPIA)	(SPIL)	(SPRA)	(TC)	(TP)
BRUSS2D-MIX	$N = 1000$	13,6	377,4	45,4	118,7	357,3	107,1	16,2	41,6	14,7	411,6	48,7
BRUSS2D-ROW	$N = 1000$	5,1	274,4	37,4	131,6	266,8	124,7	10,8	21,5	6,4	299,2	42,2
EMEP		2,4	26,9	4,2	4,1	25,4	4,1	2,5	5,9	2,4	29,8	4,1
MEDAKZO	$N = 200$	1,8	211,0	17,3	20,1	209,8	19,5	17,7	16,0	6,5	222,8	15,8
MEDAKZO	$N = 800$	0,8	227,1	15,7	20,6	225,9	19,3	14,9	15,6	4,7	236,1	15,0
MEDAKZO	$N = 2400$	−0,5	223,8	15,7	18,3	229,9	17,9	14,7	15,5	6,3	251,3	18,1
STARS-CON	$N = 1000$	0,0	0,4	0,0	0,1	0,5	0,1	0,1	0,1	0,0	0,6	0,1
STARS-MIX	$N = 1000$	0,0	0,4	0,0	0,0	0,4	0,0	0,0	0,1	0,0	0,5	0,0

Tab. 3.8: Sequentieller Overhead der Lastbalancierungimplementierungen und der Implementierung (D) auf einem Knoten des Supercomputers JUMP in %.

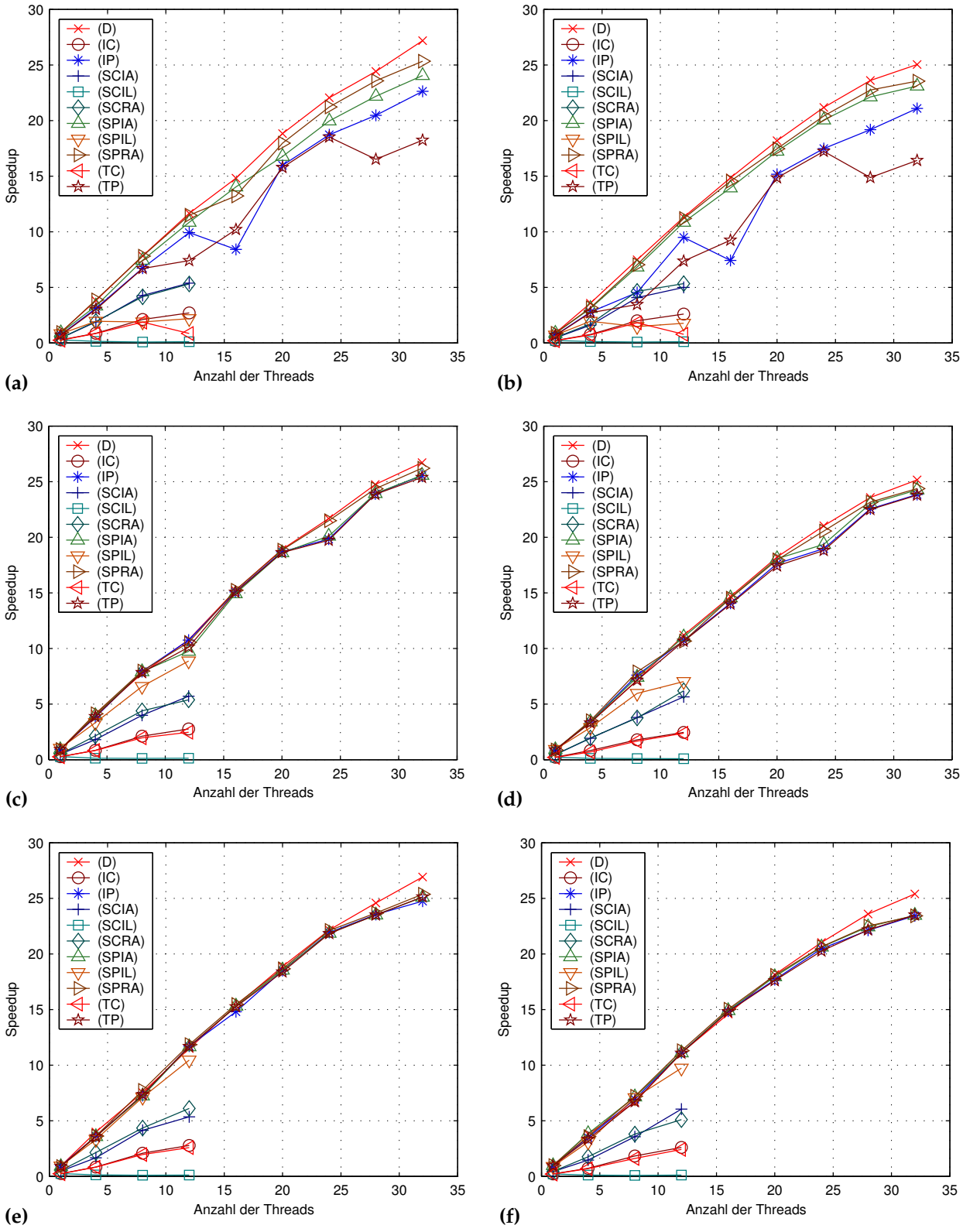


Abb. 3.23: Speedups der Lastbalancierungsimplementierungen für das Testproblem BRUSS2D mit $N = 1000$ für verschiedene Vorgaben der Zeilengröße im Vergleich zu Implementierung (D) gemessen auf einem Knoten des Supercomputers JUMP. **(a)** BRUSS2D-ROW, Zeilengröße 128 Byte, **(b)** BRUSS2D-MIX, Zeilengröße 128 Byte, **(c)** BRUSS2D-ROW, Zeilengröße 512 Byte, **(d)** BRUSS2D-MIX, Zeilengröße 512 Byte, **(e)** BRUSS2D-ROW, Zeilengröße 1024 Byte, **(f)** BRUSS2D-MIX, Zeilengröße 1024 Byte.

Die höchsten Speedup-Werte wurden für die Lastbalancierungsimplementierungen bei Vorgabe einer Zeilengröße von 512 Byte erzielt. (SPRA) kann in diesem Fall für BRUSS2D-MIX und BRUSS2D-ROW einen Speedup von 24,4 bzw. 26,4 erreichen. (SPIA) ist mit Speedups von 24,2 bzw. 25,6 etwas langsamer. (IP) und (TP) verbessern sich ebenfalls und ziehen mit (SPIA) und (SPRA) nahezu gleichauf. Sie erreichen beide einen Speedup von 23,8 für BRUSS2D-MIX sowie Speedups von 25,5 bzw. 25,4 für BRUSS2D-ROW. Besonders stark profitiert (SPIL) von der Erhöhung der Zeilengröße, bleibt jedoch immer noch deutlich hinter (IP) und (TP) zurück. Wird die Vorgabe der Zeilengröße auf 1024 Byte erhöht, können bei Verwendung kleiner Prozessorzahlen, d. h. für bis zu ca. 24 Prozessoren, die Speedups verbessert bzw. zumindest aufrecht erhalten werden. Werden dagegen 28 oder 32 Prozessoren benutzt, verringert sich der Anstieg der Speedup-Kurve, und es werden geringere Speedup-Werte als bei Verwendung einer Zeilengröße von 512 Byte erreicht. Eine mögliche Ursache für diesen Effekt ist die grobere Taskgranularität, die sich um so stärker auswirkt, je geringer die Anzahl der pro Prozessor auszuwertenden Komponenten ist.

EMEP. Für EMEP (Abb. 3.24 (a)) bewegen sich die Speedup-Werte der statischen Arbeitsverteilung in einem ähnlichen Rahmen wie auf dem Opteron-SMP. Sie erreicht auch hier ihren maximalen Speedup, der rund 1,35 beträgt, bei Verwendung von drei Prozessoren. Allerdings kann dieser Speedup durch Lastbalancierung kaum verbessert werden. Nur (TP) ist als einzige Implementierung in der Lage, einen marginal besseren Speedup bei Verwendung von zwei Prozessoren zu erzielen, der jedoch ebenfalls lediglich rund 1,35 beträgt. Ungewöhnlich an diesem Sachverhalt ist, daß in allen zuvor betrachteten Experimenten (IP) gleiche oder bessere Speedups als (TP) liefern konnte, da (IP) die gleiche Funktionalität wie (TP) realisiert, jedoch einen geringeren sequentiellen Overhead besitzt und beliebig viele Tasks in konstanter Zeit stehlen kann, während für (TP) die Kosten des Stehlens mit der Anzahl der Tasks ansteigen.

Ein ähnliches Verhalten wurde bereits in früheren Untersuchungen von Lastbalancierungsverfahren für verschiedene irreguläre Anwendungen beobachtet (z. B. Korch u. Rauber 2004b). Die Ursache besteht oft in der Verringerung der Zugriffsfrequenz auf einen oder mehrere Locks aufgrund des höheren Overheads. Zur Verdeutlichung zeigt Abb. 3.25, wie sich für ein synthetisches Testprogramm die Laufzeit eines Master-Threads verändert, der während seiner Ausführung mehrfach einen Lock sperrt und wieder freigibt, wenn die Anzahl der konkurrierend auf diesen Lock zugreifenden Arbeitsthreads – und damit die Zugriffsfrequenz auf diesen Lock – erhöht wird. Wie dieses Beispiel zeigt, wächst die Laufzeit, die der Master-Thread für das Sperren und Freigeben des Locks benötigt, stärker als linear mit der Anzahl der Arbeitsthreads. Bei feingranularen Anwendungen, deren Laufzeit durch Wartezeiten aufgrund des Sperrens von Locks dominiert wird, kann deshalb eine geringfügige Erhöhung des Overheads dazu führen, daß sich durch die geringere Zugriffsfrequenz die Wartezeiten für das Sperren von Locks signifikant verbessern und schließlich eine geringere Laufzeit resultiert.

MEDAKZO. Die auf dem IBM-p690-SMP für das Testproblem MEDAKZO mit der Problemgröße $N = 2400$ gemessenen Speedup-Werte zeigt Abb. 3.24 (b). Für dieses Beispiel kann die statische Arbeitsverteilung mit deutlichem Abstand die geringste Laufzeit erreichen. Sie ermöglicht einen maximalen Speedup von 3,23 bei Verwendung von fünf Prozessoren. Die schnellste Lastbalancierungsimplementierung, (SPRA), erreicht dagegen nur einen Speedup von 2,88, der ebenfalls bei Verwendung von 5 Prozessoren erzielt wird. Die Implementierung (SPIA), die sich von (SPRA) nur durch die Strategie für das Wechseln der Thread-ID unterscheidet, erreicht nur einen Speedup von 2,68. Das Vorgehen zum Wechseln der Thread-ID übt also in diesem Experiment einen wesentlichen Einfluß auf die Laufzeit aus. (IP) und (TP) sind mit einem Speedup von 2,61 bzw. 2,62 nur wenig langsamer als (SPIA). Auch im Vergleich von (SCRA) und (SCIA) ist die randomisierte Strategie für den Thread-ID-Wechsel mit einem maximalen Speedup von 2,47 gegenüber 2,44 effizienter, der Unterschied ist jedoch geringer als im Vergleich von (SPRA) und (SPIA). Die lockbasierte Variante der Strategie „Simple“, (SPIL), erreicht nur einen maximalen Speedup von 2,06. Am schlechtesten skalieren die Implementierungen (IC), (TC) und (SPIL), u. a. weil sie einen besonders hohen sequentiellen Overhead besitzen. (TC) und (IC) verbessern zwar ihre Laufzeit mit wachsender Prozessorzahl für bis zu 5 bzw. 6 Prozessoren, jedoch sind sie auch in diesem Fall kaum schneller als die sequentielle Referenzimplementierung. (SCIL) kann nicht von einer parallelen Ausführung profitieren.

STARS. Die Abb. 3.24 (c) und 3.24 (d) zeigen die für die beiden Anordnungen der Komponenten, STARS-CON und STARS-MIX, gemessenen Speedup-Werte. Wie bereits auf dem Itanium-2- und dem Opteron-

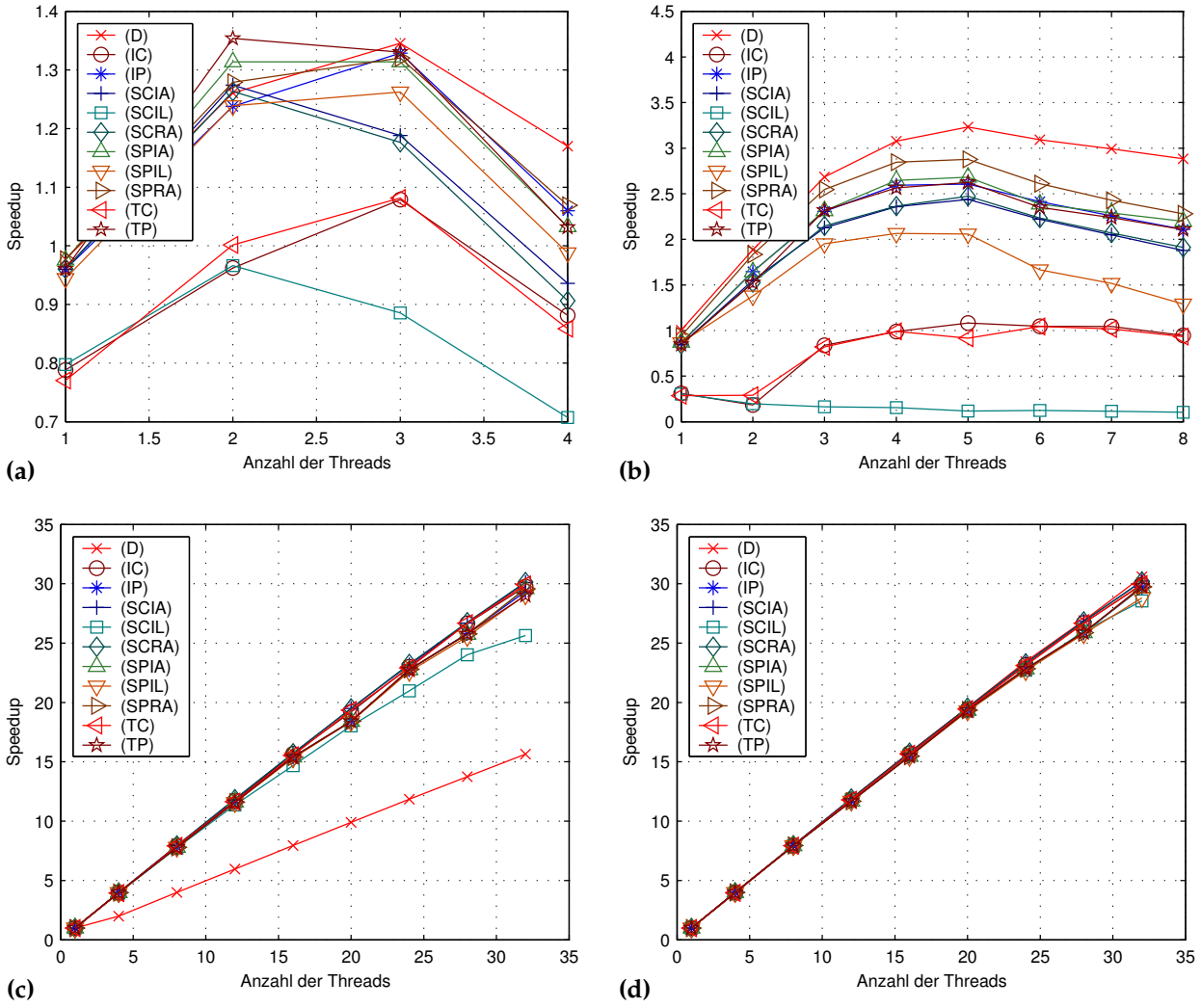


Abb. 3.24: Speedups der Lastbalancierungsimplementierungen im Vergleich zu Implementierung (D) gemessen auf einem Knoten des Supercomputers JUMP. **(a)** EMEP, **(b)** MEDAKZO, $N = 2400$, **(c)** STARS-CON, $N = 1000$, **(d)** STARS-MIX, $N = 1000$.

SMP erreichen alle Implementierungen aufgrund der hohen Taskgranularität sehr gute Speedup-Werte. Eine Ausnahme stellt lediglich die statische Arbeitsverteilung bei Anwendung auf das Testproblem STARS-CON dar, da hier eine stark unausgeglichene Anfangsverteilung vorliegt. Für die statische Arbeitsverteilung kann hier deshalb nur ein maximaler Speedup von 15,7 erzielt werden. Bei einer ausgeglichenen Anfangsverteilung, wie sie für STARS-MIX vorliegt, erreicht die statische Arbeitsverteilung dagegen unter allen Implementierungen den besten Speedup, der 30,6 beträgt.

Für STARS-MIX sind die schnellsten Lastbalancierungsimplementierungen (SCIA) und (SCRA), die beide einen Speedup von 30,2 erreichen. Die Speedups von (IC), (IP), (SPIA), (SPRA), (TC) und (TP) liegen zwischen 29,7 und 30,0. (SCIL) und (SPIL) sind mit Speedups von 28,6 bzw. 28,8 am langsamsten. Für STARS-CON liefern ebenfalls (SCIA) und (SCRA) den höchsten Speedup-Wert von 30,1. Anders als für STARS-MIX können sich (TC) und (IC) mit einem Speedup von 29,9 bzw. 29,8 von den übrigen Implementierungen geringfügig absetzen, denn (SPIA) und (SPRA) erreichen nur einen Speedup von 29,6, und für (IP), (TP) und (SPIL) wurden nur Speedups von 29,3, 29,0 und 29,1 gemessen. Die schlechteste Performance liefert wiederum (SCIL) mit einem maximalen Speedup von 25,6.

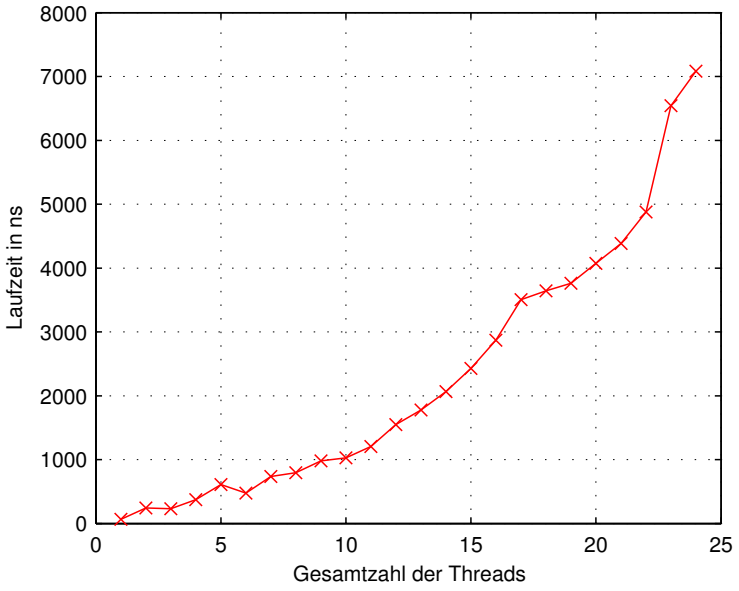


Abb. 3.25: Skalierbarkeitsverhalten eines auf Basis von Compare & Swap realisierten Spin-Locks auf einer Sun Fire 6800 mit 24 Prozessoren. Der Lock-Mechanismus wurde von Hoffmann (2003) implementiert und mit Hilfe eines Testprogramms untersucht. Gemessen wurde dabei die Laufzeit, die ein Master-Thread im Mittel benötigt, um den Lock zu sperren und ihn sofort danach wieder freizugeben, wobei parallel dazu eine vorgegebene Anzahl von Arbeitsthreads ebenfalls versucht, den Lock zu sperren und freizugeben. Die Arbeitsthreads führen im Gegensatz zum Master-Thread einige Operationen innerhalb und außerhalb des durch den Lock geschützten Bereiches aus, die die Entnahme und Verarbeitung eines Tasks simulieren.

Xeon-SMP

Als letzte Zielplattform betrachten wir ein SMP-System, bestehend aus vier Xeon-Prozessoren mit einer Taktfrequenz von 2,2 GHz. Eine Besonderheit dieses Systems besteht darin, daß die Prozessoren Hyper-Threading unterstützen und somit bereits hardwareseitig die Ausführung zweier Threads ermöglichen. Wie auf dem Opteron-SMP wird `pthread_spinlock_t` für die Realisierung der lockbasierten Implementierungen verwendet, und Fetch & Inc wird mittels Compare & Swap emuliert. Die erzeugten Threads werden jeweils an einen logischen Prozessor gebunden. Dies ist auf diesem System für das Erreichen einer hohen Performance besonders wichtig, denn nur so können die gestarteten Threads so auf die physikalischen Prozessoren verteilt werden, daß diese effizient genutzt werden können. Beispielsweise ist es für eine bestmögliche Ausnutzung der verfügbaren Rechenleistung erforderlich, im Fall, daß maximal so viele Threads gestartet werden wie physikalische Prozessoren vorhanden sind, jeden Thread auf einem separaten physikalischen Prozessor auszuführen. Übersteigt die Anzahl der Threads die Anzahl der physikalischen Prozessoren, erfolgt die Zuordnung so, daß eine Gleichverteilung der Threads auf die physikalischen und logischen Prozessoren angestrebt wird und nur solche Threads dem gleichen physikalischen Prozessor zugeordnet werden, die benachbarte Datenbereiche verarbeiten, um eine gute Ausnutzung der Lokalität zu ermöglichen.

Sequentieller Overhead. Da die Architektur des Xeon-Prozessors der Architektur des Opteron-Prozessors in vielen Punkten ähnelt, liegt auch der sequentielle Overhead der Implementierungen (Tab. 3.9) in der gleichen Größenordnung. Insbesondere gelten qualitativ die gleichen Aussagen im Vergleich der Implementierungen untereinander und im Vergleich zu den übrigen betrachteten Zielplattformen.

BRUSS2D. Die für BRUSS2D mit $N = 1000$ gemessenen Speedup-Werte sind in den Abb. 3.26 (a) und 3.26 (b) dargestellt. Die höchsten Speedups erreicht die statische Arbeitsverteilung. Für BRUSS2D-ROW wurde ihr maximaler Speedup von 2.25 bei Verwendung von drei Threads gemessen. Für BRUSS2D-MIX

Testproblem	Problemgröße	(D)	(IC)	(IP)	(SCIA)	(SCIL)	(SCRA)	(SPIA)	(SPIL)	(SPRA)	(TC)	(TP)
BRUSS2D-MIX	$N = 250$	1,6	56,3	8,2	52,9	57,1	46,5	10,1	12,2	9,1	62,7	9,1
BRUSS2D-MIX	$N = 1000$	−1,0	53,1	5,1	49,9	54,7	42,9	7,6	8,5	6,6	61,7	5,9
BRUSS2D-ROW	$N = 250$	−0,3	52,7	5,9	52,3	52,0	42,7	9,0	8,3	6,1	57,8	7,2
BRUSS2D-ROW	$N = 1000$	−2,2	52,6	4,8	51,3	50,1	42,1	6,9	7,0	6,0	58,8	5,7
EMEP		1,0	6,0	2,3	7,1	5,8	6,3	2,3	2,4	2,5	7,0	2,2
MEDAKZO	$N = 200$	−0,8	36,6	6,2	29,6	36,2	22,5	6,3	7,5	6,2	36,1	6,3
MEDAKZO	$N = 800$	−1,4	44,9	6,6	35,7	44,5	25,5	6,6	9,4	6,7	42,7	6,9
MEDAKZO	$N = 2400$	−3,3	40,0	4,3	34,5	44,0	24,1	5,2	6,1	4,7	42,8	5,4
STARS-CON	$N = 1000$	0,0	0,8	0,3	0,7	0,6	0,8	0,3	0,2	0,2	0,5	0,2
STARS-MIX	$N = 1000$	0,0	0,2	−0,2	0,2	0,5	0,1	0,0	−0,2	0,1	0,5	−0,2

Tab. 3.9: Sequentieller Overhead der Lastbalancierungimplementierungen und der Implementierung (D) auf einem Vier-Wege-Xeon-SMP in %.

beträgt ihr maximaler Speedup 2,28, der ebenfalls bei Verwendung von drei Threads gemessen wurde. Diese Werte sind etwa mit den auf dem Opteron-SMP gemessenen Werten von 2,10 bzw. 2,26 vergleichbar. Die besten Lastbalancierungsimplementierungen sind (IP) und (TP). Sie erreichen für BRUSS2D-ROW bei Verwendung von vier Threads ihre maximalen Speedups von 2,23 bzw. 2,18. Für BRUSS2D-MIX erreichen sie ihre maximalen Speedups von 2,14 bzw. 2,19 ebenfalls bei Verwendung von vier Threads. (IC) erreicht ähnlich gute Speedups von 2,15 für BRUSS2D-ROW und 2,16 für BRUSS2D-MIX, die bei Verwendung von 6 Threads erreicht werden. Diese Implementierung kann also geringfügig vom HyperThreading profitieren. Implementierungen, die besonders stark vom HyperThreading profitieren und ihren maximalen Speepdup bei Verwendung von acht Threads erreichen, sind (SPIA), (SPRA), (SCIA), (SCRA) und (SPIL), die für BRUSS2D-ROW maximale Speedups von 2,06, 2,07, 1,78, 1,73 und 1,63 sowie für BRUSS2D-MIX maximale Speedups von 2,05, 2,09, 1,75, 1,74 und 1,60 erzielen. Die Implementierung (TC) erreicht ihre maximalen Speedups von 1,82 für BRUSS2D-ROW bzw. 1,81 für BRUSS2D-MIX bei Verwendung von vier Threads. Die parallelen Laufzeiten von (SCIL) liegen, wie bereits auf anderen Maschinen, noch unterhalb ihrer sequentiellen Laufzeit.

EMEP. Die unterschiedlich hohen Funktionsauswertungszeiten der ODE-Komponenten im Fall von EMEP erlauben es einigen Lastbalancierungsimplementierungen, höhere Speedups zu erzielen als die statische Arbeitsverteilung (siehe Abb. 3.26 (c)). Im Vergleich zu den anderen betrachteten Zielplattformen wurden auf dem Xeon-SMP für EMEP die höchsten Speedup-Werte gemessen. Allerdings können auch hier nur zwei bzw. drei Prozessoren effizient genutzt werden. Bei Verwendung von zwei Threads liefern alle Lastbalancierungsimplementierungen einen besseren Speedup, der zwischen 1,36 und 1,43 liegt, als die statische Arbeitsverteilung, die in diesem Fall nur einen Speedup von 1,33 erreicht. Wird die Threadanzahl auf drei erhöht, kann die statische Arbeitsverteilung jedoch ihren Speedup auf 1,47 steigern, während (IC), (TC), (SCIA), (SCRA) und (SCIL) ihren Speedup nicht auf diesen Wert steigern können. (IP), (TP), (SPIA), (SPRA) und (SPIL) können jedoch ihren Speedup auf einen Wert zwischen 1,53 und 1,57 verbessern, wobei (SPIA) und (SPRA) den höchsten Speedup von 1,57 erreichen. Es sind also alle die Implementierungen erfolgreich, deren Arbeitseinheiten sich an der Cachezeilengröße orientieren, also eine höhere Taskgranularität besitzen.

MEDAKZO. Für MEDAKZO mit $N = 2400$ erreicht die statische Arbeitsverteilung mit deutlichem Abstand die höchsten Speedups (siehe Abb. 3.26 (d)). Bereits bei Verwendung von vier Threads erzielt sie einen Speedup von 3,36, der von keiner der Lastbalancierungsimplementierungen erreicht wird. Bei Verwendung von 8 Threads steigert sich ihr Speedup sogar auf 3,80. Der höchste Speedup einer Lastbalancierungsimplementierung beträgt 3,23 und wurde für (TP) bei Verwendung von sieben Threads gemessen. Der Speedup von (IP) liegt für vier bis acht Threads relativ konstant bei rund 2,9. Wie bereits für BRUSS2D profitieren (SPIA), (SPRA) und (SPIL) vom HyperThreading und erreichen ihre maximalen Speedups von 3,03, 2,99 und 2,65 bei Verwendung von acht Threads. Unter den feingranularen Implementierungen kann nur (IC) ihren Speedup bei Erhöhung der Threadanzahl von eins bis auf sieben stetig steigern und dadurch einen Speedup von 2,30 erreichen. (TC) erreicht ihren maximalen Speedup von 1,66 dagegen bereits bei Verwendung von drei Threads. (SCIA), (SCRA) können sogar nur für bis zu zwei Threads ihren Speedup verbessern, und (SCIL) ist bei einer parallelen Ausführung immer langsamer als bei Ausführung auf nur einem Prozessor.

STARS. Wie bereits auf den anderen Rechnersystemen ermöglicht auch hier die hohe Taskgranularität das Erreichen sehr guter Speedup-Werte. Abbildung 3.26 (e) und 3.26 (f) zeigen die für STARS-CON bzw. STARS-MIX ermittelten Meßwerte. Für das Testproblem STARS-MIX trifft diese Aussage aufgrund der ausgeglichenen Anfangsverteilung auf alle Implementierungen einschließlich der statischen Arbeitsverteilung zu. Ihren maximalen Speedup erreichen alle Implementierungen für STARS-MIX bei Verwendung von vier Threads. Werden zusätzliche Threads gestartet, führt der zusätzliche Overhead zu einer Verlangsamung. Für STARS-CON treffen diese Aussagen analog zu, allerdings mit dem Unterschied, daß sich die statische Arbeitsverteilung aufgrund der unausgeglichenen Anfangsverteilung anders verhält. Da für sie keine Möglichkeit besteht, die ungleiche Anfangsverteilung durch eine Umverteilung auszugleichen, erreicht sie bei Anwendung auf STARS-CON und Verwendung von vier Threads nur einen Speedup von 2,010. Bei Erhöhung der Threadanzahl bis auf sechs kann dieser noch bis auf 3,030 verbessert werden, da in diesem Fall

die Zuordnung der Threads zu den physikalischen Prozessoren zu einer gleichmäßigeren Lastverteilung auf die physikalischen Prozessoren führt.

Die statische Arbeitsverteilung erreicht für STARS-MIX einen maximalen Speedup von 3,954. Alle Lastbalancierungsimplementierungen erbringen demgegenüber bessere Speedups zwischen 3,963 und 3,996. Dabei sind (SCIA), (SCRA) und (SCIL) mit Speedups kleiner als 3,970 langsamer als die übrigen Lastbalancierungsimplementierungen. Etwas schneller sind (IC) und (TC) Speedups von 3,975 und 3,979. Die Implementierungen auf Cachezeilenbasis können jedoch die höchsten Speedups erzielen. Unter diesen ist (SPIL) mit einem Speedup von 3,985 am langsamsten. Für (SPRA), (IP) und (SPIA) wurden Speedups von 3,988, 3,992 und 3,995 gemessen. Der höchste gemessene Speedup beträgt 3,996 und wurde von (TP) erzielt.

Für STARS-CON sind die langsamsten Lastbalancierungsimplementierungen wiederum (SCIL), (SCIA) und (SCRA) mit maximalen Speedups von 3,882, 3,930 und 3,923. Anders als für STARS-MIX ist jedoch (SPIL) mit einem Speedup von 3,944 am langsamsten. (SPIA) und (SPRA) sind mit Speedups von 3,951 und 3,953 jedoch nur geringfügig schneller. Die Strategien „Task Queue“ und „Interval Queue“ zeigen sich hier der Strategie „Simple“ überlegen, denn die höchsten Speedups werden von (IC), (TC), (IP) und (TP) erzielt, wobei (IC) und (TC) mit Speedups von 3,987 und 3,981 etwas langsamer sind als (IP) und (TP) mit Speedups von 3,995 und 3,991.

Zusammenfassung und Schlußfolgerungen

Es wurden Laufzeitexperimente auf vier SMP-Systemen mit unterschiedlicher Architektur durchgeführt. Auf allen Systemen zeigen die Ergebnisse der Experimente, daß die Lastbalancierungsimplementierungen für bestimmte Testprobleme erfolgreicher sein können als eine statische Arbeitsverteilung. Dies ist insbesondere dann der Fall, wenn sich die Funktionsauswertungszeiten der Komponenten des zu integrierenden gewöhnlichen Differentialgleichungssystems stark voneinander unterscheiden, wie beispielsweise für die Testprobleme EMEP und STARS-CON. Doch auch für feingranulare Probleme mit einer gleichmäßigeren anfänglichen Lastverteilung können mit Hilfe von Lastbalancierungstechniken in bestimmten Fällen geringere Laufzeiten erzielt werden. Voraussetzung dafür ist jedoch, daß es gelingt, die Lastbalancierung so zu realisieren, daß nur ein sehr geringer Overhead entsteht. Ist dies der Fall, können Lastbalancierungsimplementierungen auch für feingranulare, ausgeglichene Probleme mindestens eine nur unwesentlich geringere Laufzeit als eine statische Arbeitsverteilung erreichen. Liegt jedoch eine weniger ausgeglichene Anfangsverteilung vor oder führen andere Einflüsse zu einer asynchronen Arbeitsweise der Threads, besitzen Lastbalancierungsimplementierungen einen Vorteil gegenüber einer statischen Arbeitsverteilung.

In den durchgeführten Experimenten waren die Lastbalancierungsimplementierungen auf einem Itanium-2-SMP besonders erfolgreich. Auf dieser Maschine konnten zur Realisierung der lockbasierten Implementierungen effiziente Mutex-Variablen des Typs `pthread_spinlock_t` genutzt werden. Weiterhin war mit Hilfe der atomaren Maschineninstruktion Fetch & Add eine effiziente Realisierung der Lastbalancierungsstrategie „Simple“ möglich. Die schlechtesten Ergebnisse im Vergleich zu einer statischen Arbeitsverteilung wurden auf einem Knoten des Supercomputers JUMP, einem IBM-p690-SMP, erzielt. Hier wurden Mutex-Variablen des Typs `pthread_mutex_t` zur Realisierung der lockbasierten Implementierungen verwendet. Für die Implementierung der Strategie „Simple“ wurde auf die Kernelfunktion `fetch_and_add()` zurückgegriffen. Der daraus resultierende hohe Overhead verhinderte, daß die Lastbalancierungsimplementierungen für die Testprobleme BRUSS2D, MEDAKZO und STARS-MIX einen höheren Speedup ereichen konnten als eine statische Arbeitsverteilung. Der Rückstand der jeweils besten Lastbalancierungsimplementierung auf die statische Arbeitsverteilung ist jedoch hinreichend gering, um allein durch den Einsatz effizienterer Synchronisationsmechanismen auf Basis der Maschineninstruktionen Load & Reserve und Store Conditional analog Hoffmann u. a. (2004b) eine wesentliche Verbesserung dieser Situation erwarten zu können.

Der Vergleich der untersuchten Lastbalancierungsimplementierungen erlaubt es nicht, eine Lastbalancierungsstrategie zu identifizieren, die unter allen Umständen zu einer geringeren Laufzeit führt als andere Strategien. Vielmehr hängt der Erfolg einer Lastbalancierungsstretegie von mehreren Faktoren ab, darunter in erster Linie die Eigenschaften des Differentialgleichungssystems, speziell die Verteilung und die Granularität der Funktionsauswertungszeiten. Aber auch die Hardwarearchitektur der Zielplattform, insbesondere die Verfügbarkeit geeigneter effizienter Synchronisationsmechanismen, spielt eine wichtige Rolle. Es läßt sich jedoch anhand der Ergebnisse der durchgeführten Experimente folgendes feststellen:

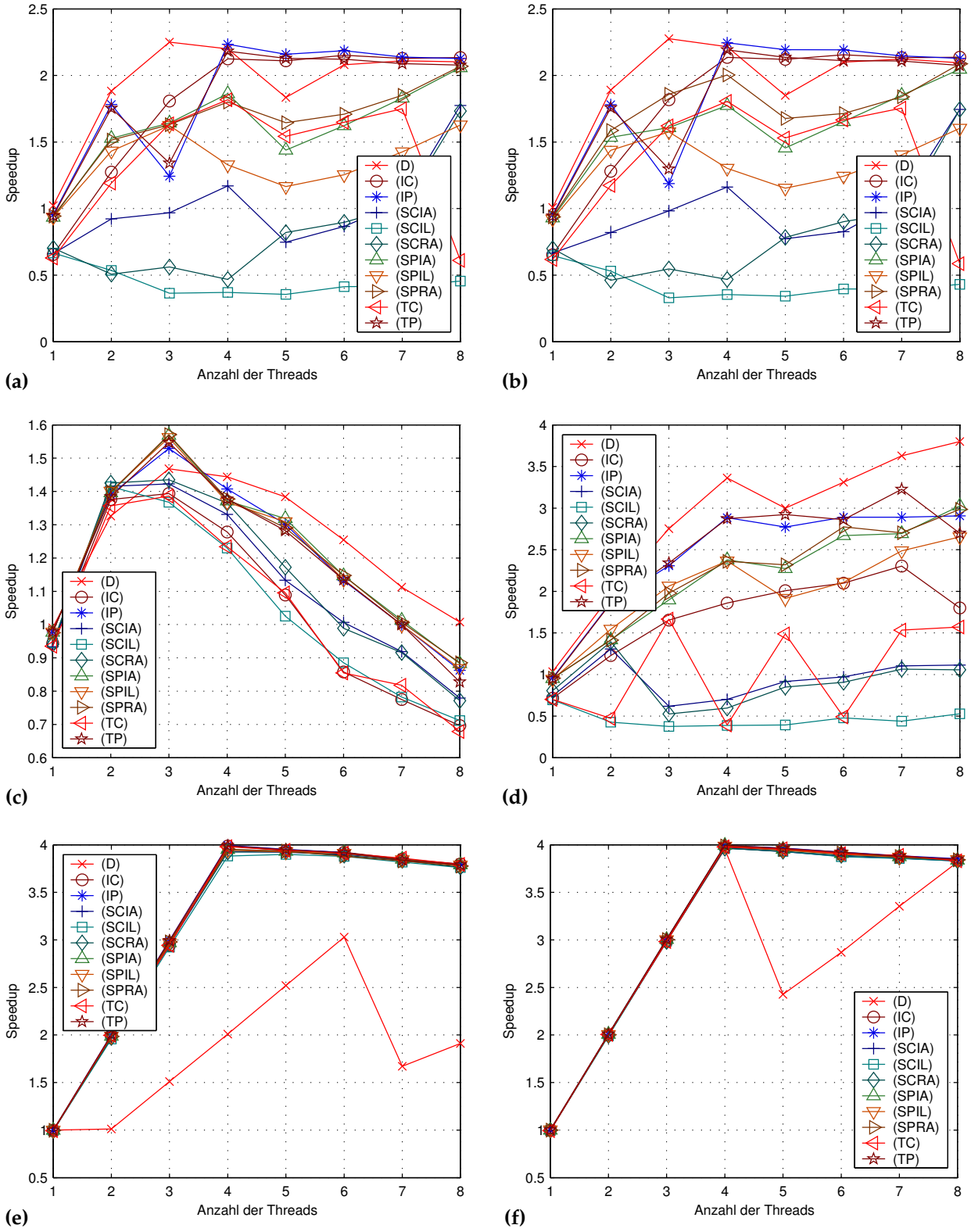


Abb. 3.26: Speedups der Lastbalancierungsimplementierungen im Vergleich zu Implementierung (D) gemessen auf einem Vier-Wege-Xeon-SMP. **(a)** BRUSS2D-ROW, $N = 1000$, **(b)** BRUSS2D-MIX, $N = 1000$, **(c)** EMEP, **(d)** MEDAKZO, $N = 2400$, **(e)** STARS-CON, $N = 1000$, **(f)** STARS-MIX, $N = 1000$.

- Die Strategie „Simple“ kann trotz ihrer einfachen Arbeitsweise sehr effizient arbeiten, wenn geeignete Maschineninstruktionen (z. B. eine atomare Fetch-&-Add-Instruktion) zur Verfügung stehen. In diesem Fall erfordert sie nur einen sehr geringen sequentiellen Overhead, und auch bei einer parallelen Ausführung entstehen Wartezeiten nur durch die Latenz der Fetch-&-Add-Instruktion. Aus diesem Grund eignet sich die Strategie „Simple“ insbesondere für feingranulare Probleme, deren Anfangsverteilung nur ein geringfügiges Lastungleichgewicht impliziert. Sie hat allerdings den Nachteil, daß bei einer ungünstigen Anfangsverteilung der Fall eintreten kann, daß viele Threads auf dem gleichen Datenbereich arbeiten und auf den gleichen Zähler zugreifen. Dies verschlechtert das Lokalitätsverhalten und erhöht die Latenz der Fetch-&-Add-Instruktion.

- Für die beiden Varianten der Strategie „Simple“, die unterschiedliche Strategien zum Wechseln der Thread-ID realisieren, wurden in vielen Experimenten ungefähr gleiche Speedups gemessen. Unter den durchgeführten Experimenten gibt es jedoch auch sowohl solche, in denen die Strategie „Increment“ signifikant höhere Speedups erreicht, als auch solche, in denen die Strategie „Random“ einen besseren Speedup liefert. Erklärbar ist dies dadurch, daß die Strategie „Random“ zwar einerseits die Wahrscheinlichkeit reduzieren kann, daß viele Threads gleichzeitig die gleiche Thread-ID besitzen, dies andererseits jedoch zu einem schlechteren Lokalitätsverhalten führt. Die Auswertung der Experimente, die auf dem IBM-p690-SMP durchgeführt wurden, welcher unter den betrachteten Systemen die größte Prozessoranzahl besitzt, deutet jedoch darauf hin, daß die Strategie „Random“ eine bessere Skalierbarkeit aufweist als die Strategie „Increment“. Denn für eine große Anzahl von Threads ist bei Verwendung der Strategie „Increment“ die Gefahr größer, daß viele Threads zur gleichen Zeit die gleiche Thread-ID annehmen.

- Die Strategie „Task Queue“ bietet die Möglichkeit, Tasks in beliebiger Reihenfolge in Warteschlangen einzufügen. Um ein gutes Lokalitätsverhalten zu gewährleisten, wird von dieser Möglichkeit jedoch kein Gebrauch gemacht. Stattdessen werden in den Warteschlangen immer zusammenhängende Intervalle von Schleifeniterationen gespeichert. Die Strategie „Interval Queue“ bietet genau diese eingeschränkte Funktionalität, nutzt jedoch die Einschränkung auf zusammenhängende Intervalle zur Reduzierung des Overheads aus. Speziell ist sie in der Lage, das Stehlen mehrerer Tasks in konstanter Zeit zu realisieren, während für die Strategie „Task Queue“ diese Kosten linear mit der Anzahl der zu stehlenden Tasks ansteigen. Prinzipiell ist daher der Strategie „Interval Queue“ ein höheres Skalierbarkeitspotential zuzuordnen. In den durchgeführten Experimenten erreichen die beiden Strategien im Rahmen der Meßgenauigkeit oft ungefähr gleiche Speedups. Dies hängt damit zusammen, daß – mit Ausnahme von EMEP und STARS-CON – die betrachteten Testprobleme bereits eine relativ gut ausgeglichene Anfangsverteilung besitzen und deshalb nur wenige Taskmigrationen durchgeführt werden müssen. In anderen Fällen kann es dazu kommen, daß sich der höhere Overhead der Strategie „Task Queue“ positiv auf die Wartezeiten bezüglich der Locks auswirkt, da er die Zugriffsfrequenz auf die Locks reduziert. Einige Beispiele, speziell im Vergleich der Implementierungsvarianten, die einzelne Schleifeniterationen als Tasks verwenden, machen jedoch das größere Potential der Strategie „Interval Queue“ deutlich.

- Die untersuchten Implementierungen der Strategien „Task Queue“ und „Interval Queue“ benutzen Locks zum Schutz der Warteschlangen, um zeitkritische Abläufe zu vermeiden. Je nach Verfügbarkeit werden dazu die im POSIX-Standard definierten Funktionen `pthread_spinlock_t` oder `pthread_mutex_t` genutzt. Im Zusammenhang mit den zur Manipulation der Warteschlangen notwendigen Operationen führt dies zu einem sequentiellen Overhead, der den der Strategie „Simple“ übersteigt. Bei einer parallelen Ausführung besitzen die Strategien „Task Queue“ und „Interval Queue“ jedoch den Vorteil, daß Wartezeiten beim Sperren von Locks nur während des Stehlens von Tasks auftreten und Tasks weniger häufig gestohlen werden als bei Verwendung der Strategie „Simple“. Darüber hinaus verfügen sie durch die Verarbeitung zusammenhängender Intervalle über ein besseres Lokalitätsverhalten. Sie sind deshalb besonders gut für Probleme mit einer sehr unausgeglichenen Anfangsverteilung geeignet, die eine effiziente Migrationsstrategie erfordern.

- Für viele feingranulare Probleme ist es vorteilhaft, mehrere benachbarte Schleifeniterationen zu einem Task zusammenzufassen und so die Taskgranularität zu erhöhen. Dies reduziert den sequentiellen Overhead und kann auch zu einer Verringerung der parallelen Overheads führen, da die Ausführungsfrequenz von Synchronisationsoperationen reduziert wird und gegebenenfalls weniger Tasks migriert werden müssen. Das Zusammenfassen mehrer Iterationen hat darüber hinaus den Vorteil, daß sich die An-

zahl der Cache-Fehlzugriffe aufgrund von False Sharing reduzieren läßt, wenn man die Anzahl der zusammenzufassenden Iterationen so auswählt und die Datenstrukturen derart ausrichtet, daß die während der Ausführung eines Tasks zugegriffenen Datenblöcke mit ein oder mehreren vollständig ausgenutzten Cachezeilen korrespondieren. Dies wirkt sich insbesondere auf die Implementierungen der Strategie „Simple“ positiv aus, da hier häufig Tasks, deren Datenbereiche räumlich benachbart sind, auf unterschiedlichen Prozessoren ausgeführt werden. Wird die Granularität jedoch zu hoch gewählt, kann dies zu einem nicht balancierbaren Lastungleichgewicht führen.

Insgesamt betrachtet, konnten durch den Einsatz von Lastbalancierungsverfahren – außer für das sehr grobgranulare Testproblem STARS-CON – in den meisten Experimenten nur geringe oder gar keine Verbesserungen des Speedups gegenüber der statischen Arbeitsverteilung erzielt werden. Dem gegenüber steht, daß die Speedups vieler feingranularen Testprobleme z. T. sehr sensibel auf Änderungen der Systemgröße reagieren. Dies legt nahe, daß die Skalierbarkeit überwiegend durch Wartezeiten für Speicherzugriffsoperationen, also vor allem die Speicherbandbreite auf Hardwareseite und die Speicherzugriffslokalität auf Programmseite, bestimmt wird.

Dennoch erscheint eine weitere Erforschung der Einsatzmöglichkeiten von Lastbalancierungsverfahren anstrebenswert. Die betrachteten Implementierungen bieten viele Optimierungsmöglichkeiten, beispielsweise den Austausch der vom Betriebssystem bzw. einer Standardbibliothek zur Verfügung gestellten Mutex-Variablen durch handoptimierte, in Assemblersprache realisierte Spin-Locks oder den Übergang zu nichtblockierenden und wartezeitfreien Datenstrukturen. Weiterhin sind eine Reihe von Modifikationen der Lastbalancierungsstrategien denkbar sowie die Anwendung alternativer Strategien. Möglichkeiten zur Bestimmung einer ausgeglichenen Anfangsverteilung anhand von zur Laufzeit ermittelten Informationen über die Funktionsauswertungszeiten sowie Techniken zur dynamischen Anpassung der Taskgranularität wurden ebenfalls bisher noch nicht untersucht. Da in der Mehrzahl der durchgeführten Experimente die statische Arbeitsverteilung entweder langsamer oder nur geringfügig schneller als die jeweils beste Lastbalancierungsimplementierung war, könnte die Entwicklung effizienterer Lastbalancierungsstrategien universell einsetzbare Implementierungen ermöglichen, die für eine große Problemklasse mindestens nahezu die gleiche Laufzeit wie die statische Arbeitsverteilung erreichen, jedoch im Falle eines Lastungleichgewichts dieses ausgleichen und die Laufzeit verbessern können.

3.7 Zusammenfassung

Das größte Parallelitätspotential eingebetteter Runge-Kutta-Verfahren liegt in der Parallelität bezüglich des gewöhnlichen Differentialgleichungssystems, da dieses Potential mit zunehmender Systemgröße wächst. Das Parallelitätspotential bezüglich der Stufen ist dagegen begrenzt und erfordert Einschränkungen in bezug auf die verwendbaren Verfahrenskoeffizienten. Eine parallele Ausführung von Zeitschritten ist aufgrund der Abhängigkeiten zwischen den Zeitschritten im allgemeinen nicht möglich. Es liegt daher nahe, Möglichkeiten zur praktischen Ausnutzung der Systemparallelität zu untersuchen.

Die Parallelität bezüglich des gewöhnlichen Differentialgleichungssystems zu nutzen, bedeutet, die Schleifen über die Systemdimension zu parallelisieren, indem die Iterationen dieser Schleifen auf die beteiligten Kontrollflüsse verteilt werden. Unabhängig von der Speicherarchitektur des Zielsystems hat dabei eine blockweise Verteilung der Iterationen eine Reihe von Vorteilen. Auf Rechnern mit verteiltem Adreßraum vereinfacht sie den Datenaustausch, und auf Rechnern mit gemeinsamem Adreßraum verbessert sie das Lokalitätsverhalten.

Zwar sind die Iterationen der Schleifen über die Systemdimension unabhängig voneinander, die Datenabhängigkeiten der Funktionsauswertungen und der Schrittweitenkontrolle erfordern jedoch eine geeignete Synchronisation der Kontrollflüsse. So muß in jeder Stufe sichergestellt werden, daß die für die Funktionsauswertungen benötigten Komponenten des jeweiligen Argumentvektors verfügbar sind. Ein verteilter Adreßraum erfordert dazu einen Nachrichtenaustausch, der z. B. bei Verwendung von MPI mit Hilfe der Funktion `MPI_Allgather()` durchgeführt werden kann. Werden bei einer Implementierung für gemeinsamen Adreßraum die Argumentvektoren als gemeinsame Datenstrukturen genutzt, genügt hier die Ausführung einer Barrier-Operation, um sicherzustellen, daß kein Kontrollfluß mit der Ausführung von Funktionsauswertungen beginnt, bevor alle anderen Kontrollflüsse die Berechnung der ihnen zugeordneten Teile des Argumentvektors abgeschlossen haben. Bestimmte Implementierungsvarianten für ge-

meinsamen Adreßraum können darüber hinaus die Ausführung einer zweiten Barrier-Operation pro Stufe erfordern, um zu verhindern, daß Teile des Argumentvektors überschrieben werden, bevor alle Funktionsauswertungen ausgeführt wurden.

Zur Durchführung der Schrittweitenkontrolle wird eine Approximation des lokalen Fehlers benötigt, die häufig als Maximumsnorm eines (skalierten) Fehlervektors bestimmt wird. Da die Elemente des Fehlervektors aufgrund der Parallelisierung bezüglich des Systems verteilt auf die beteiligten Kontrollflüsse vorliegen, ist hierzu ein Datenaustausch notwendig. Bei einer Implementierung für verteilten Adreßraum mit Hilfe von MPI kann dafür die Operation `MPI_Allreduce()` genutzt werden. Bei Verwendung von POSIX Threads zur Erstellung von Implementierungen für gemeinsamen Adreßraum steht keine derartige Operation zur Verfügung. Für kleine Prozessorzahlen ist jedoch oft eine sequentielle Bestimmung des Maximums effizient, die z. B. durch geeignete Barrier-Synchronisation realisiert werden kann.

Die Skalierbarkeit auf Parallelrechnern mit verteiltem Adreßraum wird in der Regel durch das Laufzeitverhalten der Kommunikationsoperationen bestimmt, da die Geschwindigkeit des Verbindungsnetzwerks zwischen den Knoten oft wesentlich geringer ist als die Geschwindigkeit der Hauptspeicherzugriffe und die Arbeitsgeschwindigkeit der Prozessoren. Die theoretische Analyse des Skalierbarkeitsverhaltens zeigt, daß bei wachsender Prozessorzahl vor allem die Allreduce-Operation die asymptotische Laufzeit dominiert. Ein gutes Skalierbarkeitsverhalten ergibt sich jedoch, wenn die Problemgröße im gleichen Verhältnis wie die Prozessorzahl gesteigert wird. Dies ist insbesondere dann der Fall, wenn die Funktionsauswertungszeit mit wachsender Problemgröße ansteigt.

Die Synchronisation der Kontrollflüsse auf Rechnern mit gemeinsamem Adreßraum erfolgt im wesentlichen mit Hilfe von Barrier-Operationen. Zwar steigt auch die Laufzeit dieser Operationen mit wachsender Prozessorzahl an, es erfolgt jedoch kein Datenaustausch, so daß ihre Laufzeit wesentlich geringer als die der MPI-Operationen ist. Die Programmlaufzeit und die Skalierbarkeit wird daher wesentlich durch die Laufzeit der Speicherzugriffsoperationen bestimmt, die wiederum abhängig von der Geschwindigkeit des Speichersystems und dem Lokalitätsverhalten ist. Die Analyse der Arbeitsräume der in den betrachteten Implementierungsvarianten enthaltenen Schleifen zeigt, daß die größten Arbeitsräume – diejenigen der Schleifen über die Systemdimension – durch die Parallelisierung indirekt proportional zur Prozessorzahl kleiner werden. Dadurch sind in bestimmten Fällen superlineare Speedups möglich. Eine Ausnahme bilden jedoch Schleifen, in denen Funktionsauswertungen ausgeführt werden, da diese auch Komponenten des Argumentvektors außerhalb des dem jeweiligen Kontrollfluß zugeordneten Indexbereichs benutzen können. Dies betrifft insbesondere Implementierungen, bei denen alle Argumentvektoren getrennt gespeichert werden. Für Probleme mit kleiner Zugriffsdistanz und hoher Systemgröße hat dieser Effekt jedoch keine wesentlichen Nachteile für die getrennte Speicherung der Argumentvektoren zur Folge.

Zur Untersuchung des Skalierbarkeitsverhaltens auf realen Rechnersystemen wurden Laufzeitexperimente mit auf MPI basierenden Implementierungen für verteilten Adreßraum und mit auf POSIX Threads basierenden Implementierungen für gemeinsamen Adreßraum sowie mit hybriden Implementierungen, die beide Programmiermodelle kombinieren, durchgeführt. Die MPI-Implementierungen zeigten dabei auf der überwiegenden Zahl der betrachteten Rechnersysteme keine zufriedenstellende Skalierbarkeit. Insbesondere auf aus Standardkomponenten aufgebauten Clustersystemen, deren Knoten über Infiniband oder Gigabit bzw. Fast Ethernet vernetzt waren, wurden sehr schlechte Speedup-Werte gemessen. Doch auch auf Parallelrechnern mit gemeinsamem Adreßraum ist die Skalierbarkeit der MPI-Implementierungen nicht überzeugend. Auf einem Knoten des Supercomputers JUMP konnte beispielsweise nicht einmal ein Speedup von 4 erreicht werden. Demgegenüber konnten die POSIX-Threads-Implementierungen in vergleichbaren Experimenten auf dem selben 32-Prozessor-System Speedups bis zu einem Wert von 23,0 erzielen. Eine Implementierung konnte für kleine Prozessorzahlen sogar superlineare Speedups erreichen. Dennoch entspricht der für 32 Prozessoren erzielte Speedup nur einer Effizient von 0,72 und ist somit noch weit vom optimalen Wert entfernt. Hybride Implementierungen, die MPI und POSIX Threads kombinieren, können auf SMP-Clustersystemen eine bessere Skalierbarkeit erreichen als reine MPI-Implementierungen. Da sich die Anzahl der erzeugten MPI-Prozesse und die Anzahl der erzeugten Threads so wählen lassen, daß die hybriden Implementierungen wie reine MPI- oder POSIX-Threads-Implementierungen arbeiten, liefern sie auf Parallelrechnern mit uniformer Speicherarchitektur nahezu die gleiche Performance wie die entsprechenden spezialisierten Implementierungen.

Da allerdings auch die Implementierungen für gemeinsamen Adreßraum nicht alle CPUs moderner Parallelrechner optimal ausnutzen können, erscheint es notwendig, nach weiteren Möglichkeiten zur Ver-

besserung der Skalierbarkeit zu suchen. Eine mögliche Ursache für eine nichtoptimale Skalierbarkeit kann in einem Lastungleichgewicht liegen. Dieses kann beispielsweise durch eine ungleichmäßige Verteilung der Funktionsauswertungszeiten über die Komponenten des Differentialgleichungssystems, durch unterschiedliche Prozessorgeschwindigkeiten oder durch zur Laufzeit auftretende Ereignisse hervorgerufen werden. Um ein solches Lastungleichgewicht auszugleichen, können dynamische Lastbalancierungsverfahren eingesetzt werden. Aufgrund der oft sehr feinen Granularität und der oft geringen Funktionsauswertungszeiten besteht die Schwierigkeit dabei darin, die Lastbalancierung mit möglichst geringem Overhead zu realisieren, so daß der durch die Lastbalancierung erzielte Laufzeitgewinn diesen Overhead mehr als ausgleichen kann. Als Zielarchitektur eignen sich deshalb vor allem Parallelrechner mit gemeinsamem Adreßraum, da hier über den gemeinsamen Speicher ein effizienterer Datenaustausch möglich ist und spezielle Maschineninstruktionen zur Synchronisation der Kontrollflüsse genutzt werden können. Dabei muß jedoch auch dafür gesorgt werden, daß trotz dynamischer Arbeitsverteilung ein möglichst gutes Lokalitätsverhalten bewahrt wird. Laufzeitexperimente mit verschiedenen Lastbalancierungsstrategien auf Parallelrechnern mit gemeinsamem Adreßraum mit unterschiedlicher Architektur zeigen, daß durch den Einsatz dynamischer Lastbalancierung in bestimmten Fällen sogar für Probleme mit ausgewogener Verteilung der Funktionsauswertungszeiten ein signifikanter Laufzeitgewinn erzielt werden kann. Die eingesetzten Implementierungen wurden allerdings nur mit dem Ziel entwickelt, die Einsetzbarkeit dieses Ansatzes zu zeigen, und lassen Raum für Verbesserungen. Es ist daher davon auszugehen, daß mit Hilfe verbesserter Implementierungen höhere Laufzeitgewinne möglich sind.

4 Gleichungen mit beschränkter Zugriffsdistanz

Bei allen bisherigen Betrachtungen waren die Transformationsmöglichkeiten zur Optimierung des Programmcodes durch die Annahme eingeschränkt, daß die Funktionsauswertungen der Komponenten der rechten Seite des gewöhnlichen Differentialgleichungssystems beliebige Komponenten des Argumentvektors benutzen dürfen. Dies ist einer der entscheidenden Gründe dafür, warum die bisher untersuchten Implementierungsvarianten noch nicht die gewünschte Effizienz erreichen. Dieses Kapitel akzeptiert deshalb eine problemspezifische Spezialisierung durch die Annahme einer beschränkten Zugriffsdistanz, und es diskutiert neue Optimierungsmöglichkeiten, die daraus hervorgehen.

4.1 Spezialisierung auf Gleichungen mit beschränkter Zugriffsdistanz

Die in den vorhergehenden Kapiteln durchgeführten Untersuchungen haben gezeigt, wie wichtig ein gutes Lokalitätsverhalten für die effiziente Ausführung eingebetteter Runge-Kutta-Verfahren ist. Sie haben jedoch auch deutlich gemacht, daß im allgemeinen Fall Datenabhängigkeiten die Transformationsmöglichkeiten einschränken, wodurch alle bisher betrachteten Implementierungsvarianten nicht die gewünschte Effizienz erreichen können. Das größte Hindernis ist dabei die sequentielle Abhängigkeit der Stufen. Sie führt dazu, daß das Lokalitätsverhalten überwiegend durch die Wiederverwendung vollständiger Vektoren bestimmt wird. Bei einer Parallelisierung verbessert sich diese Situation zwar, dennoch überschreiten die resultierenden Arbeitsmengen für größere Probleme oft die Cachegröße. Weiterhin erzwingt die sequentielle Abhängigkeit der Stufen eine aufwendige Kommunikation bzw. Synchronisation der beteiligten Kontrollflüsse. Zur Verbesserung der Effizienz eingebetteter Runge-Kutta-Verfahren erscheint es daher notwendig, die sequentielle Abhängigkeit der Stufen aufzubrechen, um weiterführende Programmtransformationen zu ermöglichen. Da dies im allgemeinen Fall aufgrund der vorhandenen Datenabhängigkeiten der Funktionsauswertungen nicht möglich ist, erfordert dies jedoch eine Spezialisierung entweder bezüglich der Verfahrenskoeffizienten oder bezüglich des gewöhnlichen Differentialgleichungssystems.

Die erste Möglichkeit dazu wäre die Verwendung von Verfahrenskoeffizienten, die eine Entkoppelung der Stufen ermöglichen, wie sie bereits von verschiedenen Autoren zum Zwecke der Parallelisierung untersucht wurden (vgl. Abschnitt 1.2.1). Dies würde allerdings Einschränkungen bezüglich der numerischen Eigenschaften des Verfahrens nach sich ziehen und dennoch nicht zu einer vollständigen Entkoppelung aller Stufen führen. In diesem Kapitel wird deshalb ein alternativer Ansatz vorgeschlagen, der auf einer Spezialisierung bezüglich des zu integrierenden Problems beruht. Bereits Rauber u. Rünger (2001) erkannten, daß sich das spezifische Zugriffsmuster der rechten Seite des Testproblems BRUSS2D-MIX für Lokalitätsoptimierungen ausnutzen läßt, und skizzierten einen als „Pipelining-Ansatz“ bezeichneten sequentiellen Algorithmus mit dem Potential zur Reduzierung der im Cache zu speichernden Datenmenge. Im folgenden wird diese Idee aufgegriffen, formalisiert und verallgemeinert. Es werden konkrete sequentielle und parallele Implementierungsvarianten präsentiert, ausführlich bezüglich ihres Lokalitätsverhaltens und ihrer Skalierbarkeit untersucht und miteinander verglichen. Weiterhin wird ein Ansatz zur Reduzierung des Berechnungsaufwandes durch eine Modifikation der Schrittweitenkontrolle vorgeschlagen, und es werden Einsatzmöglichkeiten von Techniken zur dynamischen Lastbalancierung diskutiert.

Bereits bei der theoretischen Analyse der Arbeitsräume war die Berücksichtigung der Zugriffsdistanz hilfreich, um eine präzisere Abschätzung der Arbeitsräume zu ermöglichen. Wie sich dabei herausstellte, bestimmt die Zugriffsdistanz die Größe einiger Arbeitsräume und besitzt dadurch einen entscheidenden

Einfluß auf das Lokalitätsverhalten. Dies gilt insbesondere für Implementierungen, die alle Argumentvektoren als separate Datenfelder speichern. Es liegt daher nahe, die Zugriffsdistanz bei der Suche nach Möglichkeiten zur Optimierung des Lokalitätsverhaltens gesondert zu berücksichtigen. Wir gehen deshalb im folgenden davon aus, daß die Zugriffsdistanz der Funktion $\mathbf{f}$ bekannt ist und für Optimierungen genutzt werden kann.

Ausgangspunkt aller nachfolgenden Betrachtungen ist die Annahme, daß die Funktion $\mathbf{f}$, die die rechte Seite des gewöhnlichen Differentialgleichungssystems beschreibt, eine *beschränkte Zugriffsdistanz* besitzt:

Definition 4.1 (Beschränkte Zugriffsdistanz)**.** Die Zugriffsdistanz $d(\mathbf{f})$ einer Funktion $\mathbf{f}(t, \mathbf{w})$ mit $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ wird als *beschränkt* bezeichnet, wenn gilt: $d(\mathbf{f}) \ll n$.

Nach Definition 2.1 bezeichnet die Zugriffsdistanz $d(f_j)$ einer Komponentenfunktion $f_j(t, \mathbf{w})$ den kleinsten Wert b, so daß f_j nur die Teilmenge

$$\{w_{j-b}, w_{j-b+1}, \ldots, w_j, \ldots, w_{j+b-1}, w_{j+b}\}$$

der Komponenten des Argumentvektors $\mathbf{w}$ benutzt. Das bedeutet, $f_j(t, \mathbf{w})$ ist unabhängig von w_i für $|i - j| > b$, und infolgedessen gilt für diesen Fall $\frac{\partial f_j(t,\mathbf{w})}{\partial w_i} = 0$. Als Konsequenz davon besitzt die Jacobi-Matrix $\mathbf{f}_\mathbf{w}(t, \mathbf{w})$ der Funktion $\mathbf{f}$, definiert als

$$\mathbf{f}_\mathbf{w}(t, \mathbf{w}) = \begin{pmatrix} \frac{\partial f_1(t,\mathbf{w})}{\partial w_1} & \cdots & \frac{\partial f_1(t,\mathbf{w})}{\partial w_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n(t,\mathbf{w})}{\partial w_1} & \cdots & \frac{\partial f_n(t,\mathbf{w})}{\partial w_n} \end{pmatrix} , \tag{4.1}$$

eine Bandstruktur mit Halbbandbreite $d(\mathbf{f}) + 1$ und Bandbreite $D = 2d(\mathbf{f}) + 1$. Da auch $\frac{\partial f_j(t,\mathbf{w})}{\partial w_i} = 0$ für $t_0 \le t \le t_e$ impliziert, daß $f_j(t, \mathbf{w})$ unabhängig von w_i ist, läßt sich umgekehrt auch aus der Beschränktheit der Bandbreite der Jacobi-Matrix auf die Beschränktheit der Zugriffsdistanz schließen. Dies bedeutet insbesondere, daß alle in diesem Kapitel unter Annahme einer beschränkten Zugriffsdistanz entwickelten Optimierungen genau auf diejenige Klasse von Anfangswertproblemen angewendet werden können, die eine Jacobi-Matrix mit Bandstruktur besitzen. Typisch ist dies – eine geeignete Anordnung der ODE-Komponenten vorausgesetzt – u. a. für Probleme, die durch eine Ortsdiskretisierung mittels Linienmethode aus partiellen Differentialgleichungssystemen entstehen, wie z. B. BRUSS2D-MIX und MEDAKZO.

Die Bedeutung der Zugriffsdistanz für die Datenabhängigkeiten zwischen den Stufen liegt darin, daß eine *Lokalität der Datenabhängigkeiten* gegeben ist, d. h., daß jede Komponentenfunktion f_j jeweils nur wenige Komponenten des Argumentvektors benutzt, die zu einem schmalen, zusammenhängenden Indexbereich gehören, dem auch der Index j angehört. Welche genauen Auswirkungen dies hat, wird deutlich, wenn man betrachtet, wie sich die Abhängigkeiten über mehrere Stufen hinweg fortpflanzen. Wir betrachten dazu, von welchen Komponenten des Argumentvektors $\mathbf{w}_1$ die Funktionsauswertung $f_j(t_\kappa + c_s h_\kappa, \mathbf{w}_s)$ des Argumentvektors $\mathbf{w}_s$ abhängig ist.[1] Dazu sei $d = d(\mathbf{f})$ die maximale Zugriffsdistanz der Funktion $\mathbf{f}$. Nach Definition 2.1 greift die Funktionsauswertung $f_j(t_\kappa + c_s h_\kappa, \mathbf{w}_s)$ zur Berechnung der Stufenvektorkomponente v_{sj} höchstens auf die Komponenten

$$\{w_{s,j-d}, \ldots, w_{s,j+d}\}$$

des Argumentvektors $\mathbf{w}_s$ zu. Um diese zu berechnen, werden die Stufenvektorkomponenten

$$\{v_{l,j-d}, \ldots, v_{l,j+d}\} \quad \text{für} \quad l = 1, \ldots, s-1$$

benötigt, die aus den Funktionsauswertungen

$$\{f_{j-d}(t_\kappa + c_l h_\kappa, \mathbf{w}_l), \ldots, f_{j+d}(t_\kappa + c_l h_\kappa, \mathbf{w}_l)\} \quad \text{für} \quad l = 1, \ldots, s-1$$

[1] Analog könnte man bestimmen, welche Funktionsauswertungen $f_j(t_\kappa + c_s h_\kappa, \mathbf{w}_s)$ des Argumentvektors $\mathbf{w}_s$ von einer gegebenen Komponente w_{1i} des Argumentvektors $\mathbf{w}_1$ abhängig sind.

resultieren. Diese wiederum benötigen insgesamt höchstens die Komponenten

$$\{w_{l,j-2d},\ldots,w_{l,j+2d}\} \quad \text{für} \quad l=1,\ldots,s-1$$

der Argumentvektoren $\mathbf{w}_1,\ldots,\mathbf{w}_{s-1}$, zu deren Berechnung die Stufenvektorkomponenten

$$\{v_{l,j-2d},\ldots,v_{l,j+2d}\}=\{f_{j-2d}(t_\kappa+c_l h_\kappa,\mathbf{w}_l),\ldots,f_{j+2d}(t_\kappa+c_l h_\kappa,\mathbf{w}_l)\} \quad \text{für} \quad l=1,\ldots,s-2$$

der Stufenvektoren $\mathbf{v}_1,\ldots,\mathbf{v}_{s-2}$ erforderlich sind. Es wird deutlich, daß sich bei einem Übergang von einer Stufe l auf eine Stufe $l-1$ der Indexbereich der höchstens benötigten Argument- und Stufenvektorkomponenten an beiden Enden um d, also insgesamt um $2d$ vergrößert. Führt man diese Überlegungen induktiv fort, ergibt sich, daß zur Durchführung der Funktionsauswertung $f_j(t,\mathbf{w}_s)$ höchstens die Komponenten

$$\{w_{l,j-(s-l+1)d},\ldots,w_{l,j+(s-l+1)d}\} \quad \text{für} \quad l=1,\ldots,s$$

der Argumentvektoren $\mathbf{w}_1,\ldots,\mathbf{w}_s$ und die Komponenten

$$\{v_{l,j-(s-l+1)d},\ldots,v_{l,j+(s-l+1)d}\} \quad \text{für} \quad l=1,\ldots,s-1$$

der Stufenvektoren $\mathbf{v}_1,\ldots,\mathbf{v}_{s-1}$ zuvor bekannt sein müssen. Die Antwort auf die Ausgangsfrage lautet also, daß die Funktionsauswertung $f_j(t_\kappa+c_s h_\kappa,\mathbf{w}_s)$ von den $2sd+1$ Komponenten

$$\{w_{1,j-sd},\ldots,w_{1,j+sd}\}$$

des Argumentvektors $\mathbf{w}_1$ abhängt.

Aus diesen Betrachtungen ergibt sich als wichtigste Folgerung, daß zur Berechnung einer Komponente j der Approximationsvektoren $\eta_{\kappa+1}$ und $\hat{\eta}_{\kappa+1}$, ausgehend von dem Approximationsvektor des vorherigen Zeitschrittes, $\eta_\kappa=\mathbf{w}_1$, kein Argument- bzw. Stufenvektor vollständig bestimmt werden muß. Vielmehr genügt es, wenn jeweils nur ein zusammenhängender, j einschließender Indexbereich dieser Vektoren bekannt ist, dessen Größe abhängig von der Zugriffsdistanz der Funktion $\mathbf{f}$ und der Nummer der jeweiligen Stufe ist. Somit ist es nicht mehr erforderlich, die Stufen strikt nacheinander zu berechnen, sondern es ergeben sich verschiedene Transformations- und Partitionierungsmöglichkeiten, die zur Optimierung des Lokalitätsverhaltens und zur Parallelisierung ausgenutzt werden können.

4.2 Lokalitätsoptimierungen für sequentielle Implementierungen

Als erstes soll versucht werden, die Transformationsmöglichkeiten, die sich durch die Annahme einer beschränkten Zugriffsdistanz ergeben, zur Verbesserung des Lokalitätsverhaltens zu nutzen.

Die Berechnungskernel der allgemeinen Implementierungen besitzen eine Schleifenstruktur mit einer maximalen Schachtelungstiefe von 3.[2] Dabei ist allen Implementierungen gemeinsam, daß die äußerste Schleife über alle Stufen iteriert, da die Stufen aufgrund der Datenabhängigkeiten nacheinander berechnet werden mußten. Dies führt dazu, daß die Größen der wichtigsten Arbeitsmengen Vielfache der Systemgröße sind. Die Schleifen der beiden übrigen Schachtelungsebenen iterieren entweder über die Systemdimension oder über eine Teilmenge der Stufen. Diese beiden Schachtelungsebenen können ausgetauscht oder durch Tiling ineinander verschachtelt werden.

Der Ansatz zur Verkleinerung der Arbeitsräume besteht nun darin, die Schleifenstruktur so zu verändern, daß die Schleife über die Systemdimension zur äußersten Schleife wird. Dadurch würde sich der Arbeitsraum der äußersten Schleife wesentlich verringern, da die inneren Schleifen nun ausschließlich über Stufen iterieren und somit der Arbeitsraum asymptotisch von der Stufenzahl s statt von der Systemdimension n abhängen würde. Durch die Beschränkung der Zugriffsdistanz wurden die Stufen jedoch nicht vollständig entkoppelt. Da wir damit rechnen müssen, daß eine Funktionsauswertung $f_j(t,\mathbf{w}_l)$ auf bis zu $2d(\mathbf{f})+1$ Komponenten von $\mathbf{w}_l$ zugreift, müssen zuerst diese Komponenten von $\mathbf{w}_l$ bestimmt werden, bevor $f_j(t,\mathbf{w}_l)$ ausgeführt werden kann. Eine Ausnahme bilden Funktionsauswertungen für Komponenten am Anfang bzw. am Ende der Vektoren, d. h. mit Index $j\le d(\mathbf{f})$ bzw. $j>n-d(\mathbf{f})$, die auf entsprechend

[2] Wir zählen dabei durch Tiling erzeugte Schachtelungstiefen nicht mit.

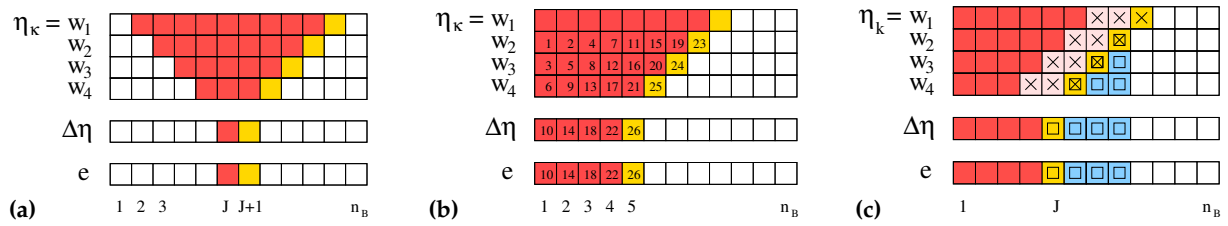


Abb. 4.1: (a) Veranschaulichung der zur Berechnung von Block J und der anschließenden Berechnung von Block $J+1$ der Vektoren $\Delta\eta$ und $\mathbf{e}$ zu berücksichtigenden Datenabhängigkeiten für den Fall $s = 4$. **(b)** Illustration des Pipelining-Berechnungsschemas. Die Zahlen innerhalb der Blöcke geben die Berechnungsreihenfolge an. Nachdem Block 4 der Vektoren $\Delta\eta$ und $\mathbf{e}$ berechnet wurde, muß zur Berechnung von Block 5 nur ein zusätzlicher Block jedes Argumentvektors bekannt sein. **(c)** Illustration des Arbeitsraumes der äußersten Schleife des Pipelining-Berechnungsschemas innerhalb der Iteration zur Berechnung von Block J der Vektoren $\Delta\eta$ und $\mathbf{e}$. Alle Blöcke, auf die während dieser Iteration zugegriffen wird, sind durch Symbole markiert. Blöcke, die innerhalb von Funktionsauswertungen benötigt werden, sind dabei mit einem Kreuz gekennzeichnet, und Blöcke, die unter Verwendung von Funktionsergebnissen aktualisiert werden, sind durch ein Quadrat markiert.

weniger Argumentvektorkomponenten zugreifen. Unter Berücksichtigung dieses Sachverhalts besteht die Idee zur Realisierung eines Algorithmus, dessen äußerste Schleife über die Systemdimension iteriert, nun darin, die Berechnung nachfolgender Stufen um eine hinreichend große Anzahl von Komponenten zu verzögern, so daß – analog zur Arbeitsweise einer Pipeline – in jeder Iteration der äußersten Schleife ein neues Element jedes Stufen- bzw. Argumentvektors berechnet werden kann.

4.2.1 Realisierung eines blockbasierten Pipelining-Algorithmus

Ein solches elementweises Pipelining der Stufen- bzw. Argumentvektoren ist zwar möglich, in der Praxis hat sich jedoch gezeigt, daß eine blockweise Arbeitsweise Geschwindigkeitsvorteile bietet. Der blockweise Algorithmus ist darüber hinaus einfacher zu realisieren und zu beschreiben. Im folgenden soll deshalb eine blockweise Implementierung auf Basis der in Abschnitt 2.4 vorgestellten Implementierung (D) näher beschrieben werden.

Die resultierende Implementierung soll mit (PipeD) bezeichnet werden. Sie speichert wie Implementierung (D) nur die Argumentvektoren $\mathbf{w}_1, \ldots, \mathbf{w}_s$ (mit $\mathbf{w}_1 = \eta_\kappa$). Die Stufenvektoren $\mathbf{v}_1, \ldots, \mathbf{v}_s$ werden dagegen nicht explizit gespeichert. Wir wählen eine Aufteilung der Argumentvektoren in Blöcke der Größe B, wobei wir eine Blockgröße $B \geq d(\mathbf{f})$ wählen. Zur Vereinfachung der Beschreibung sei n ein Vielfaches von B, so daß sich eine Aufteilung der Argumentvektoren in $n_B = n/B$ Blöcke ergibt. Der im folgenden vorgestellte Algorithmus arbeitet jedoch auch in dem Fall korrekt, daß jeweils der letzte Block eines Vektors eine kleinere Größe besitzt. Die Wahl einer Blockgröße $B \geq d(\mathbf{f})$ führt dazu, daß die Funktionsauswertung für einen Block J, $f_J(t, \mathbf{w})$ mit

$$\mathbf{f}_J(t, \mathbf{w}) = (f_{(J-1)B+1}(t, \mathbf{w}), \ldots, f_{JB}(t, \mathbf{w})) \ ,$$

höchstens Komponenten der Blöcke $J-1$, J und $J+1$ des Argumentvektors $\mathbf{w}$ benutzt. Für die Funktionsauswertung eines Blocks J angewendet auf den Argumentvektor $\mathbf{w}_s$ werden also höchstens die 3 Blöcke $J-1$, J und $J+1$ von $\mathbf{w}_s$ benötigt. Zur Berechnung dieser 3 Blöcke benutzen die zugehörigen Funktionsauswertungen höchstens Komponenten der 5 Blöcke $J-2$, $J-1$, J, $J+1$ und $J+2$ von $\mathbf{w}_{s-1}$. Setzt man dies bis zur Stufe 1 fort, werden höchstens die $2s+1$ Blöcke $J-s, \ldots, J+s$ des Argumentvektors $\mathbf{w}_1$ benötigt. Diese Abhängigkeiten werden in Abb. 4.1 (a) verdeutlicht. Diese Abbildung veranschaulicht auch, daß nach der Durchführung der Funktionsauswertung für einen Block J der Stufe s zur Bestimmung der zugehörigen Blöcke von $\Delta\eta$ und $\mathbf{e}$ und der vorausgehenden Bestimmung aller dazu notwendigen Argumentvektorblöcke der vorhergehenden Stufen nur noch ein weiterer Argumentvektorblock pro Stufe berechnet werden muß, um den nächsten Block von $\Delta\eta$ und $\mathbf{e}$ mit Index $J+1$ bestimmen zu können.

Dies läßt sich ausnutzen, um folgenden Algorithmus zu realisieren, der in Abb. 4.1 (b) veranschaulicht wird. Abbildung 4.2 zeigt den Pseudocode des Berechnungskernels der resultierenden Implementierung.

1. **Initialisierung der Pipeline:** Argumentvektor $\mathbf{w}_1$ ist bereits bekannt, da er identisch mit dem Approximationsvektor des vorherigen Zeitschrittes, η_κ, ist. Es wird daher mit der Berechnung von Block 1 des Argumentvektors $\mathbf{w}_2$ begonnen, wozu lediglich die Blöcke 1 und 2 von $\mathbf{w}_1$ bekannt sein müssen. Ein Block einer höheren Stufe kann im Moment noch nicht berechnet werden. Deshalb fahren wir mit der Berechnung von Block 2 des Argumentvektors $\mathbf{w}_2$ fort, wofür auf die Blöcke 1, 2 und 3 von $\mathbf{w}_1$ zugegriffen werden muß. Jetzt können wir Block 1 von $\mathbf{w}_3$ berechnen, da hierzu nur die Blöcke 1 und 2 von $\mathbf{w}_2$ benötigt werden. Als nächstes berechnen wir Block 3 von $\mathbf{w}_2$, Block 2 von $\mathbf{w}_3$ und Block 1 von $\mathbf{w}_4$, und fahren in gleicher Weise fort, bis Block s von $\mathbf{w}_2$ und Block 2 von $\mathbf{w}_s$ berechnet wurde. Jetzt kann der erste Block von $\Delta\eta$ und $\mathbf{e}$ berechnet werden. Danach ist die Initialisierung der Pipeline abgeschlossen.

2. **Diagonaler Sweep über die Argumentvektoren:** Nach der Initialisierung der Pipeline können wir blockweise über die Systemdimension iterieren, wobei in jedem Iterationsschritt J für $J = s+1, \ldots, n_B$ in einem diagonalen Lauf über die Argumentvektoren je ein neuer Argumentvektorblock von $\mathbf{w}_2, \ldots, \mathbf{w}_s$ und ein Block der Vektoren $\Delta\eta$ und $\mathbf{e}$ berechnet wird.

3. **Finalisierung der Pipeline:** Nachdem durch den diagonalen Sweep die letzmögliche vollständige Diagonale berechnet wurde, wird die Pipeline – der bisherigen Vorgehensweise folgend – abgebaut.

4.2.2 Theoretische Analyse des Lokalitätsverhaltens

Die wesentlichen Elemente der Implementierung (PipeD), die das Lokalitätsverhalten bestimmen, sind zum einen die Makros STAGE1() und STAGE() zur Verarbeitung einzelner Blöcke, zum anderen der diagonale Lauf über die Stufen. Die beiden Makros besitzen eine Schleifenstruktur, die derjenigen der Schleifen über die Systemdimension und der darin eingeschlossenen Schleifen der Ausgangsimplementierung (D) entspricht (vgl. Abb. 2.8). Der wichtigste Unterschied besteht darin, daß die Schleifen über die Systemdimension innerhalb der Makros nicht über alle Systemkomponenten iterieren, sondern nur über die B Komponenten eines vorgegebenen Blocks. Dementsprechend sind die Arbeitsräume (Tab. 4.1) durch Vielfache von B geprägt; aufgrund der Funktionsauswertungen aber auch durch die Zugriffsdistanz der Funktion $\mathbf{f}$.

Die entscheidenden zwei Arbeitsmengen werden jedoch durch den diagonalen Lauf über die Stufen (einschließlich Initialisierung und Finalisierung der Pipeline) geprägt. Die erste entsteht durch die Menge aller während eines Zeitschrittes verwendeten Vektoren und ermöglicht die Wiederverwendung dieser Vektoren innerhalb aufeinanderfolgender Zeitschritte. Wir gehen dazu an dieser Stelle davon aus, daß wie in Implementierung (D) alle Argumentvektoren vollständig gespeichert werden. Dann beträgt diese Arbeitsmenge, wie im Fall der Implementierung (D), $(s+3)n$, was auch dem notwendigen Speicherplatzbedarf entspricht.

Die zweite wichtige Arbeitsmenge ist geprägt durch den Arbeitsraum R_I des diagonalen Laufs über die Stufen, d. h. der Schleife in Zeile 31. Die Schleifen zur Initialisierung (Zeile 25) und zur Finalisierung (Zeile 37) der Pipeline sind dieser sehr ähnlich, allerdings entfallen einige Blöcke, die außerhalb der Vektorgrenzen liegen würden. Der Arbeitsraum R_I des diagonalen Laufs über die Stufen ist deshalb von großer Bedeutung, weil er die Wiederverwendung von Daten zwischen aufeinanderfolgenden Iterationen der über die Systemdimension laufenden Schleife, den sogenannten „Pipelining-Schritten“, ermöglicht. Diese Schleife ist die äußerste Schleife des Berechnungskernels und wird sehr häufig wiederholt. Die in diesem Arbeitsraum enthaltenen Blöcke sind in Abb. 4.1 (c) veranschaulicht. Er besteht aus $3s$ Blöcken, auf die innerhalb von Funktionsauswertungen zugegriffen wird, insgesamt $2s$ Blöcken der Vektoren $\Delta\eta$ und $\mathbf{e}$ sowie zusätzlich

$$\sum_{i=3}^{s}(i-2) = \frac{1}{2}(s-1)(s-2) = \frac{1}{2}s^2 - \frac{3}{2}s + 1$$

Argumentvektorblöcken, die unter Verwendung von Ergebnissen von Funktionsauswertungen aktualisiert werden. Hinzu kommt ein Block des Vektors $\mathbf{s}$, in dem die in jedem Pipelining-Schritt berechneten Skalierungsfaktoren gespeichert werden. Insgesamt erhalten wir somit:

$$|R_I| = \left(5s + \frac{1}{2}s^2 - \frac{3}{2}s + 2\right)B = \left(\frac{1}{2}s^2 + \frac{7}{2}s + 2\right)B = \Theta(s^2 B) \quad . \tag{4.2}$$

```
1: // Makro STAGE1: Auswertung des Blocks mit Index J des Argumentvektors w_1 = η_κ

2: #define STAGE1(J)
3: {
4:     for (j := (J − 1)B + 1; j ≤ min{JB, n}; j++)
5:     {
6:         v̄ := h f_j(t + c_1 h, η);
7:         s[j] := |η[j]| + |v̄|;
8:         for (i := 2; i ≤ s; i++) w_i[j] := η[j] + a_{i1} v̄;
9:         Δη[j] := b_1 v̄;
10:        e[j] := b̃_1 v̄;
11:    }
12: }

13: // Makro STAGE: Auswertung des Blocks mit Index J des Argumentvektors w_l

14: #define STAGE(J, l)
15: {
16:    for (j := (J − 1)B + 1; j ≤ min{JB, n}; j++)
17:    {
18:        v̄ := h f_j(t + c_l h, w_l);
19:        for (i := l + 1; i ≤ s; i++) w_i[j] += a_{il} v̄;
20:        Δη[j] += b_l v̄;
21:        e[j] += b̃_l v̄;
22:    }
23: }

24: // Initialisierung der Pipeline

25: for (J := 1; J ≤ s; J++)
26: {
27:    STAGE1(J);
28:    for (l := 2; l ≤ J; l++) STAGE(J − l + 1, l);
29: }

30: // Diagonaler Lauf über die Stufen

31: for (J := s + 1; J ≤ n_B; J++)
32: {
33:    STAGE1(J);
34:    for (l := 2; l ≤ s; l++) STAGE(J − l + 1, l);
35: }

36: // Finalisierung der Pipeline

37: for (J := 2; J ≤ s; J++)
38:    for (l := J; l ≤ s; l++) STAGE(n_B + J − l, l);
```

Abb. 4.2: Sequentielle Variante der Implementierung (PipeD).

Zeile	I	$\lvert I\rvert$	R_I	R_L	$\lvert R_I\rvert$	$\lvert R_L\rvert$
4	$j=(J-1)B+1,\ldots,JB$	B	$\mathbf{w}_2[j],\ldots,\mathbf{w}_s[j],\Delta\eta[j],\mathbf{e}[j],\mathbf{s}[j]$; D Komp. von η	B Komp. von $\mathbf{w}_2,\ldots,\mathbf{w}_s,\Delta\eta,\mathbf{e},\mathbf{s}$; $B+D-1$ Komp. von η	$D+s+2$	$(s+3)B+D-1$
7	$i=2,\ldots,s$	$s-1$	$\mathbf{w}_i[j],\eta[j]$	$\mathbf{w}_2[j],\ldots,\mathbf{w}_s[j],\eta[j]$	2	s
15	$j=(J-1)B+1,\ldots,JB$	B	$\mathbf{w}_{l+1}[j],\ldots,\mathbf{w}_s[j],\Delta\eta[j],\mathbf{e}[j]$; D Komp. von $\mathbf{w}_l$	B Komp. von $\mathbf{w}_{l+1},\ldots,\mathbf{w}_s,\Delta\eta,\mathbf{e}$; $B+D-1$ Komp. von $\mathbf{w}_l$	$D+s-l+2$	$(s-l+3)B+D-1$
18	$i=l+1,\ldots,s$	$s-l$	$\mathbf{w}_i[j]$	$\mathbf{w}_{l+1}[j],\ldots,\mathbf{w}_s[j]$	1	$s-l$

Tab. 4.1: Arbeitsräume der in der sequentiellen Variante der Implementierung (PipeD) innerhalb der Makros STAGE1() und STAGE() verwendeten Schleifen.

s	$\lvert R_I \rvert / B$
1	6
2	11
3	17
4	24

s	$\lvert R_I \rvert / B$
5	32
6	41
7	51
8	62
9	74

s	$\lvert R_I \rvert / B$
10	87
11	101
12	116
13	132

Tab. 4.2: Größe des Arbeitsraumes R_I des diagonalen Laufes über die Stufen nach (4.2) in Anzahl von Blöcken für verschiedene Stufenzahlen s.

Tabelle 4.2 zeigt die resultierende Größe des Arbeitsraumes als Anzahl von Blöcken für verschiedene Stufenzahlen.

Der Arbeitsraum R_I ist also nur von der Stufenzahl s und der Blockgröße B abhängig, jedoch nicht direkt von der Systemgröße n, wie dies für Implementierung (D) der Fall war. Die Blockgröße B muß jedoch mindestens so groß wie die Zugriffsdistanz $d(\mathbf{f})$ gewählt werden, die abhängig von der Wahl der Problemgröße sein kann. Da wir annehmen, daß die Zugriffsdistanz beschränkt ist, also $d(\mathbf{f}) \ll n$, führt der vorgeschlagene Algorithmus dennoch zu einer erheblichen Reduktion des Arbeitsraumes R_I der äußersten Schleife des Berechnungskernels. Für Implementierung (D), in der die äußerste Schleife über die Stufen iterierte, war die Größe dieses Arbeitsraumes ein Vielfaches von n. Sie verringerte sich zwar in jeder Stufe um n, betrug aber in der letzten Stufe noch $3n$.

4.2.3 Alternative Implementierungsvarianten

Die in Abschnitt 4.2.1 präsentierte Pipelining-Implementierung auf Basis der Implementierung (D) stellt nur eine von vielen möglichen Implementierungsvarianten dar. Alternative Varianten ergeben sich u. a. aus folgenden Überlegungen:

- **Makros oder Unterprogramme:** Die in Abschnitt 4.2.1 vorgestellte Implementierung basiert auf der Verwendung von Makros zur Verarbeitung eines Blocks. Eine Alternative besteht in der Verwendung von Unterprogrammen (z. B. C-Funktionen). Da die Makros an den aufrufenden Programmstellen expandiert werden, kann die Verwendung von Unterprogrammaufrufen die Codegröße wesentlich verringern und die Lokalität des Programmcodes erhöhen. Dies kann insbesondere dann von Vorteil sein, wenn dadurch Interferenzen von Speicherzugriffen vermieden werden können. Ein Nachteil der Verwendung von Unterprogrammaufrufen besteht jedoch darin, daß ein Overhead durch Sprunganweisungen, die Parameterübergabe und gegebenenfalls das Sichern und Wiederherstellen von Registerinhalten entsteht. Darüber hinaus erhöht sich die Gefahr von Konfliktfehlzugriffen auf den Instruktionscache falls sich Teile der Programmcodes der Unterprogramme untereinander oder mit dem aufrufenden Programmcode im Cache überlagern.

- **Laufrichtung:** Es ist möglich, Implementierungen zu realisieren, die nicht mit dem ersten Block des Argumentvektors $\mathbf{w}_2$ beginnen und die Pipelining-Schritte in Richtung höherer Indexwerte ausführen, sondern man kann ebenso gut mit dem letzten oder einem beliebigen anderen Block von $\mathbf{w}_2$ beginnen. Für eine gute Lokalität wird man bei einer sequentiellen Implementierung jedoch in der Regel darauf verzichten, mehrere Pipelines simultan zu verwenden. Das Laufen in Richtung absteigender Indexwerte kann auf manchen Systemen von Nachteil sein, wenn die zugrundeliegende Hardware nicht in der Lage ist, bei dieser Laufrichtung ein ebenso effizientes Prefetching durchzuführen, wie beim Lauf in Richtung aufsteigender Indexwerte.

- **Schrittgröße:** Die blockweise Realisierung des Algorithmus ist nur eine mögliche Variante. Wie bereits in der Einführung von Abschnitt 4.2 erwähnt, kann z. B. auch eine elementweise arbeitende Implementierung des Pipelining-Algorithmus realisiert werden, wenn eine entsprechende Verzögerung der Pipeline-Stufen erfolgt.

- **Alternative Basisimplementierung:** Es ist nicht zwingend erforderlich, Implementierung (D) als Ausgangsimplementierung zu nutzen. Prinzipiell können auch andere Basisimplementierungen genutzt wer-

den, allerdings mit der Einschränkung, daß aufgrund des diagonalen Laufes immer alle Argumentvektoren (oder zumindest Teile aller Argumentvektoren), gespeichert werden müssen, um eine Mehrfachberechnung von Argumentvektorelementen zu vermeiden. Interessante Implementierungsvarianten sind beispielsweise ein Schleifen-Tiling ähnlich (Dblock) oder eine Kombination der Implementierungen (E) und (D), durch die sich der Arbeitsraum eines Pipelining-Schrittes für große Stufenzahlen weiter reduzieren läßt.

- **Mehrfache Pipelining-Läufe:** Da die Größe des Arbeitsraumes eines Pipelining-Schrittes quadratisch mit der Stufenzahl wächst, kann es für große Stufenzahlen sinnvoll sein, das Pipelining-Berechnungsschema nicht auf alle Stufen simultan anzuwenden. Stattdessen kann es vorteilhaft sein, zunächst nur einen Teil der Stufen einzubeziehen und die übrigen in nachfolgenden Phasen zu berechnen.

- **Pipelining über mehrere Zeitschritte:** Wird auf eine Schrittweitenkontrolle verzichtet und stattdessen eine konstante Schrittweite verwendet, kann der Pipelining-Algorithmus über mehrere Zeitschritte ausgedehnt werden. Dadurch vergrößert sich allerdings der Arbeitsraum der Pipelining-Schritte.

4.2.4 Reduzierung des Speicherplatzbedarfs

Da aktuelle Rechnersysteme z. Z. typischerweise mit mehreren Gigabyte Hauptspeicher ausgestattet sind, können mit den bisher beschriebenen Implementierungsvarianten, die einen Speicherplatzbedarf von $(s+3)n$ bzw. $(s+5)n$ besitzen, in der Regel bereits Systeme mit mehreren Millionen Komponenten integriert werden, auch wenn die Stufenanzahl hoch ist, ohne daß der Speicherplatzbedarf eine Auslagerung auf externe Speichermedien erforderlich macht. Viele Probleme besitzen jedoch eine noch größere Anzahl von Komponenten. Einen besonders hohen Speicherbedarf besitzen oft solche Probleme, die durch eine Ortsdiskretisierung aus partiellen Differentialgleichungssystemen hervorgehen. Hier kann man in der Regel die Genauigkeit der Diskretisierung durch Parameter beeinflussen. Ist eine hohe Genauigkeit der Ortsdiskretisierung erforderlich und ist das zu diskretisierende Gebiet sehr groß, ergeben sich entsprechend große gewöhnliche Differentialgleichungssysteme. Ein häufig verwendeter Diskretisierungsansatz ist z. B. die *Linienmethode*, bei der ein äquidistantes oder variables Ortsgitter als Ausgangspunkt für die Diskretisierung genutzt wird. Die Anzahl der ODE-Komponenten hängt dann von der Anzahl der Gitterpunkte ab.

Beispiel 4.1. Betrachten wir zunächst ein äquidistantes eindimensionales Ortsgitter, das eine Strecke der Länge 1 in Abschnitte der Länge $\frac{1}{N-1}$ unterteilt. Dieses besitzt N Gitterpunkte. Erhöht man die Dimension auf 2, benötigt man bereits N^2 Gitterpunkte, um ein Einheitsquadrat mit Kantenlänge 1 in Quadrate mit Kantenlänge $\frac{1}{N-1}$ zu unterteilen. Für drei Ortsdimensionen steigt die Anzahl der Gitterpunkte bei einer analogen Unterteilung sogar auf N^3. Verfeinert man also bei einem dreidimensionalen Problem die Ortsauflösung um Faktor 10, steigt die Anzahl der Gitterpunkte etwa um Faktor 1000. Wollen wir beispielsweise ein Problem auf dem Einheitswürfel mit einer Ortsauflösung von 10^{-3} lösen, benötigen wir ein Ortsgitter mit ca. 10^9 Gitterpunkten. Wäre eine Ortsauflösung von 10^{-6} erforderlich, bestünde das Diskretisierungsgitter sogar aus rund 10^{18} Punkten.

Um Speicherplatz zu sparen, kann man zunächst versuchen, die Anzahl der Stufen zu reduzieren. Man verzichtet in diesem Fall aber auf wünschenswerte numerische Eigenschaften der Verfahren, wie z. B. eine hohe Ordnung, die erst durch eine hohe Stufenzahl ermöglicht werden. Verschiedene Autoren, darunter Kennedy u. a. (2000), Calvo u. a. (2004), Berland u. a. (2004, 2005) und Ruuth (2005) schlagen deshalb spezielle explizite Verfahren vor, die mit geringerem Speicherplatzbedarf (z. B. $2n$ oder $3n$) auskommen, obwohl sie eine höhere Anzahl von Stufen und eine höhere Ordnung besitzen. Dies wird durch eine spezielle Konstruktion der Verfahrenskoeffizienten ermöglicht, so daß in jeder Stufe nur die in den zur Verfügung stehenden Vektoren gespeicherten Daten genutzt werden und die innerhalb einer Stufe neu berechneten Daten in den verfügbaren Vektoren abgelegt werden können. Dieses Vorgehen hat jedoch verschiedene Nachteile:

1. Auch wenn man den Speicherplatzbedarf von $\Theta(sn)$ auf $\Theta(n)$ reduzieren konnte, wächst er doch immer noch linear mit n. Bei einer dreidimensionalen, mittels äquidistantem Ortsgitter diskretisierten PDE würde er also mit N^3 wachsen.

2. Es können nur spezielle Klassen von Verfahrenskoeffizienten verwendet werden. Bieten diese nicht die

zur Lösung eines betrachteten Problems erforderlichen numerischen Eigenschaften, muß man auf klassische Verfahren mit höherem Speicherplatzbedarf zurückgreifen.

3. Die Einschränkung der Freiheitsgrade bei der Wahl der Verfahrenskoeffizienten hat insbesondere zur Folge, daß in der Regel eine höhere Stufenzahl benötigt wird, um eine gewünschte Ordnung zu erreichen. Dadurch erhöht sich der notwendige Berechnungsaufwand gegenüber klassischen Verfahren.

4. Die vorgeschlagenen Verfahrensklassen enthalten oft keine eingebetteten Lösungen. Es ist daher in diesen Fällen keine ebenso effiziente Schrittweitenkontrolle möglich, wie mit eingebetteten Runge-Kutta-Verfahren.

5. Verfahren, die sowohl mit wenig Speicherplatz auskommen, aber auch eingebettete Lösung besitzen, stellen zusätzliche Bedingungen an die Verfahrenskoeffizienten und erfordern daher Zugeständnisse bezüglich der erreichbaren numerischen Eigenschaften.

6. In jedem Fall erfordert die Realisierung einer Schrittweitenkontrolle für diese Verfahren zusätzlichen Speicherplatz: Zur Schätzung des Fehlers wird in der Regel ein Vektor der Dimension n benötigt, um die zweite Approximation $\hat{\eta}_{\kappa+1}$ oder alternativ den Vektor $\Delta\eta = \hat{\eta}_{\kappa+1} - \eta_{\kappa+1}$ zu speichern. Die Speicherung dieses Vektors kann entfallen, wenn die eingebettete Lösung $\hat{\eta}_{\kappa+1}$ nach Berechnung der Stufe $s-1$ vorliegt und somit ohne Zwischenspeicherung mit der nach Berechnung der Stufe s vorliegenden Lösung $\eta_{\kappa+1}$ verglichen werden kann (siehe Kennedy u. Carpenter 1994). Um den Zeitschritt wiederholen zu können, muß darüber hinaus der im vorhergehenden Zeitschritt berechnete Approximationsvektor η_κ gespeichert werden.

7. Einige vorgeschlagene Verfahren erreichen eine Reduzierung des Speicherplatzbedarfs auf $2n$ nur unter der Voraussetzung, daß bestimmte Eigenschaften des gewöhnlichen Differentialgleichungssystems ausgenutzt werden, um beispielsweise den Argumentvektor einer Funktionsauswertung mit dem Ergebnis der Funktionsauswertung überschreiben zu können. Im allgemeinen Fall, d. h. für gewöhnliche Differentialgleichungssysteme mit beliebigen Abhängigkeiten, wird ein zusätzlicher Vektor der Größe n benötigt, um das Ergebnis der Funktionsauswertung zwischenzuspeichern (Kennedy u. a. 2000).

Daher ist es erstrebenswert, neue Verfahren bzw. Algorithmen zu finden, die diese Nachteile vermeiden. Wir betrachten deshalb die Möglichkeiten, die durch die Annahme einer beschränkten Zugriffsdistanz gegeben sind, unter diesen Gesichtspunkten neu.

Eine einfache Möglichkeit zur Implementierung des vorgeschlagenen Pipelining-Berechnungsschemas speichert alle benutzten Argumentvektoren vollständig. Der Speicherplatzbedarf entspricht somit dem der Ausgangsimplementierung (D), d. h., er beträgt $(s+3)n$. Dieses Vorgehen wurde auch zur Erstellung der nachfolgend im Rahmen von Simulations- und Laufzeitexperimenten untersuchten Implementierungsvarianten umgesetzt. Es ist jedoch möglich, den Speicherplatzbedarf erheblich zu reduzieren, wenn man nur die Vektorelemente im Speicher aufbewahrt, die zur Durchführung späterer Berechnungen erforderlich sind.

Jede Implementierung muß mindestens den Approximationsvektor $\eta = \eta_\kappa$ speichern, d. h., ein Speicherplatzbedarf kleiner n ist in keinem Fall realisierbar. Wollen wir auch weiterhin eine Schrittweitenkontrolle durchführen, benötigen wir darüber hinaus den Vektor $\Delta\eta$. Andernfalls kann η_κ durch $\eta_{\kappa+1}$ überschrieben werden. Der Fehlervektor $\mathbf{e}$ und der Skalierungsvektor $\mathbf{s}$ müssen nicht vollständig gespeichert werden. Zur Durchführung der Schrittweitenkontrolle ist es ausreichend,

$$\epsilon = \frac{1}{\mathrm{TOL}} \max_{j=1,\ldots,n} \left| \frac{\mathbf{e}[j]}{\mathbf{s}[j]} \right|$$

zu kennen. Dazu genügt es, neben dem aktuellen Wert für ϵ, von jedem der beiden Vektoren nur so viele Blöcke zu speichern, wie es der Länge der Pipeline entspricht. Das heißt, es müssen für jeden der beiden Vektoren nur s Blöcke gespeichert werden. Bezüglich der Argumentvektoren genügt es, die in dem aktuellen Arbeitsraum R_I des diagonalen Laufs über die Stufen enthaltenen $\frac{1}{2}s^2 + \frac{3}{2}s - 2$ Blöcke der Argumentvektoren $\mathbf{w}_2, \ldots, \mathbf{w}_s$ zu speichern. Damit ergibt sich ein Gesamtspeicherplatzbedarf von

$$2n + \left(\frac{1}{2}s^2 + \frac{7}{2}s - 2\right) B \ .$$

Wenn keine Schrittweitenkontrolle durchgeführt werden soll, verringert sich der Speicherplatzbedarf um $n + 2sB$, da der Vektor $\Delta\eta$ und die Vektoren **e** und **s** nicht benötigt werden. Wir kommen also mit geringerem Speicherplatz aus, als die $2n$-Verfahren, wenn wir von einer beschränkten Zugriffsdistanz ausgehen. Der Speicherplatzbedarf des Pipelining-Algorithmus läßt sich noch geringfügig verbessern, wenn man als Basisimplementierung eine Kombination aus den Implementierungsvarianten (D) und (E) nutzt.

Für die als Beispiel genannte Problemklasse der mittels Linienmethode diskretisierten partiellen Differentialgleichungssysteme, ist es erforderlich, bezüglich der Anordnung der ODE-Komponenten eine Verschachtelung der abhängigen Variablen durchzuführen, um eine beschränkte Zugriffsdistanz zu erhalten. Die zur Ortsdiskretisierung eingesetzten Operatoren approximieren eine Ortsableitung in einem betrachteten Gitterpunkt, indem sie dazu Gitterpunkte aus einer begrenzten Umgebung des betrachteten Gitterpunkts heranziehen. Es ist daher möglich, eine Raumdimension auszuwählen und eine blockweise Unterteilung bezüglich dieser Raumdimension vorzunehmen.

Beispiel 4.2. Bei einer eindimensionalen PDE exisitert nur eine Raumdimension. Die N ODE-Komponenten werden daher bezüglich dieser Raumdimension geordnet. Die Zugriffsdistanz ist in diesem Fall ein konstanter Wert, der unabhängig von der Systemgröße ist. Bei einer zweidimensionalen PDE kann man das zweidimensionale Ortsgitter in mehrere Zeilen (bzw. analog in Spalten) unterteilen und diese Zeilen nacheinander im Speicher anordnen. Die Zugriffsdistanz umfaßt in diesem Fall mehrere Zeilen, d. h., sie ist beschränkt durch $O(N)$, während die Systemgröße in $O(N^2)$ liegt. Eine dreidimensionalen PDE führt zu einer ODE mit einer Systemgröße von $O(N^3)$. Analog der zeilenweisen Speicherung für zweidimensionale Probleme kann hier eine Unterteilung des Gitters in N zweidimensionale Ebenen der Größe N^2 erfolgen. Die Zugriffsdistanz beträgt dementsprechend $O(N^2)$. Analog läßt sich diese Überlegung für höherdimensionale Probleme fortsetzen.

Läßt man den Speicherplatzbedarf für den in jedem Fall benötigten Approximationsvektor η_κ außen vor, besitzen die $2n$-Verfahren also in Abhängigkeit von N für d-dimensionale PDE-Probleme einen Speicherplatzbedarf von N^d. Dem gegenüber steht ein Speicherplatzbedarf von nur $O(s^2N^{d-1})$ für die Ausnutzung der beschränkten Zugriffsdistanz durch das Pipelining-Berechnungsschema.

Beispiel 4.3. Als Beispiel betrachten wird die Integration des Testproblems BRUSS2D-MIX mit dem 7-stufigen Verfahren DOPRI5(4) bei Verwendung von $N = 10^3$. In diesem Fall beträgt die Systemgröße $n = 2N^2 = 2 \cdot 10^6$. Die Zugriffsdistanz beträgt $2N$, so daß wir diesen Wert als minimale Blockgröße B wählen können. Ein $2n$ Verfahren besitzt demzufolge einen Speicherplatzbedarf von $2n = 4 \cdot 10^6$. Der Speicherplatzbedarf des Pipelining-Algorithmus beträgt demgegenüber nur $n + \left(\frac{1}{2}s^2 + \frac{3}{2}s - 2\right) 2N = 2,066 \cdot 10^6$. Es ergibt sich also eine Speicherplatzersparnis von rund 48 %. Würde man eine Schrittweitenkontrolle nutzen, müßte ein zusätzlicher n-Vektor gespeichert werden. Der Pipelining-Algorithmus hätte dann einen Speicherplatzbedarf von $4,094 \cdot 10^6$. Im Vergleich zum $2n$-Verfahren, kann also mit einem weniger als 2,4 % höheren Speicherplatzbedarf eine effiziente Schrittweitenkontrolle realisiert werden. Für ein $2n$-Verfahren würde sich demgegenüber bei Durchführung einer Schrittweitenkontrolle der Speicherplatzbedarf durch die zusätzliche Speicherung der beiden Vektoren η_κ und $\Delta\eta$ sogar verdoppeln.

Der Pipelining-Ansatz zur Ausnutzung der beschränkten Zugriffsdistanz hat also gegenüber den von anderen Autoren vorgeschlagenen Verfahren mit einem Speicherplatzbedarf von $2n$ folgende Vorteile:

1. Signifikant geringerer Speicherplatzbedarf von $n + O(s^2B)$.
2. Es können beliebige explizite, insbesondere eingebettete Verfahren verwendet werden.
3. Da die Verfahrenskoeffizienten keinen Einschränkungen unterliegen, kann eine höhere Ordnung mit geringerer Stufenzahl erreicht werden.
4. Eine Schrittweitenkontrolle kann effizient mit Hilfe eingebetteter Lösungen realisiert werden. Die vorgeschlagenen Verfahrenskoeffizienten der $2n$-Verfahren besitzen oft keine eingebetteten Lösungen und müssen auf alternative Möglichkeiten zur Schrittweitenkontrolle, z. B. Richardson-Extrapolation, ausweichen.
5. Die Schrittweitenkontrolle kann mit einem Speicherplatzbedarf von $2n + O(s^2B)$ realisiert werden. Die $2n$-Verfahren würden ebenfalls einen zusätzlichen Speicherplatzbedarf von mindestens n, in der Regel sogar $2n$, also insgesamt einen Speicherplatzbedarf von $3n$ bzw. $4n$ benötigen.

Ein Nachteil des Pipelining-Ansatzes besteht darin, daß nur Probleme mit beschränkter Zugriffsdistanz integriert werden können. Die 2n-Verfahren lassen zwar prinzipiell beliebige Datenabhängigkeiten innerhalb der Funktionsauswertungen zu, da jedoch nur bestimmte Klassen von Verfahrenskoeffizienten eine solche Reduktion des Speicherplatzbedarfs ermöglichen, unterliegen die erreichbaren numerischen Eigenschaften starken Einschränkungen, so daß diese Verfahren ebenfalls nur für spezielle Problemklassen geeignet sind.

4.2.5 Simulationsbasierte Analyse des Lokalitätsverhaltens

Analog zur simulationsbasierten Analyse des Lokalitätsverhaltens der allgemeinen Implementierungen (Abschnitt 2.4.3) werden im folgenden verschiedene Implementierungsvarianten des Pipelining-Berechnungsschemas untereinander, aber auch mit der allgemeinen Implementierungsvariante (D) verglichen. Dabei werden folgende Varianten des Pipelining-Algorithmus betrachtet:

- **(PipeD):** Blockbasierter Pipelining-Algorithmus, wie in Abschnitt 4.2.1 beschrieben.
- **(PipeDE):** Blockbasierter Pipelining-Algorithmus ähnlich (PipeD), jedoch wurde als Basisimplementierung eine Kombination aus (D) und (E) verwendet, um den Arbeitsraum der Pipelining-Schritte für große Stufenzahlen zu verringern. Dabei werden die Argumentvektorelemente der ersten Stufen mit einer Schleifenstruktur analog Implementierung (E) durch Aufsummieren von Stufenvektorelementen berechnet. Die Argumentvektorelemente der oberen Stufen werden dagegen analog Implementierung (D) sukzessive aktualisiert.
- **(PipeDblock):** Blockbasierter Pipelining-Algorithmus ähnlich (PipeD), jedoch wurde als Basisimplementierung (Dblock) verwendet, um ein Schleifen-Tiling zu realisieren.
- **(PipeDel):** Elementbasierter Pipelining-Algorithmus auf Basis der Implementierung (D).

Als Beispiele werden, wie in Abschnitt 2.4.3, das Testproblem MEDAKZO für den Fall $N = 15\,000$ und das Testproblem BRUSS2D-MIX für den Fall $N = 250$ betrachtet. Als Verfahren werden DOPRI8(7) für MEDAKZO und DOPRI5(4) für BRUSS2D-MIX eingesetzt. Für BRUSS2D wurde $B = 2N = 500$ und für MEDAKZO wurde $B = 8$ gewählt.

Auswirkungen der Assoziativität und der Zeilengröße

Die Variation der Assoziativität führt für die Implementierung (PipeD) sowohl für MEDAKZO (Abb. 4.3 (a)) als auch für BRUSS2D-MIX (Abb. 4.3 (b)) zu den gleichen Aussagen, wie sie bereits für die allgemeinen Implementierungen getroffen werden konnten. Die Anzahl der Cache-Fehlzugriffe verringert sich, wenn die Assoziativität vergrößert wird. Eine Erhöhung über den Wert 4 hinaus ermöglicht jedoch nur noch geringfügige Verbesserungen. Die negativen Auswirkungen einer geringen Assoziativität sind für große Caches weniger gravierend als für geringe Cachegrößen.

Wie es bereits für die allgemeinen Implementierungen der Fall war, führt auch für die Implementierung (PipeD) die Veränderung der Zeilengröße (Abb. 4.4) zu einer signifikanten Veränderung der Anzahl der Cache-Fehlzugriffe. Auch hier verringert sich prinzipiell die Anzahl der Fehlzugriffe bei Erhöhung der Zeilengröße. Allerdings vergrößern sich dadurch auch einige Arbeitsmengen, so daß es für bestimmte Cachegrößen zu einer Erhöhung der Anzahl der Fehlzugriffe kommt.

Vergleich der Implementierungen und Identifikation der Arbeitsmengen

Die Abbildungen 4.5 und 4.6 zeigen einen Vergleich verschiedener Implementierungsvarianten für die Testprobleme MEDAKZO und BRUSS2D-MIX bei Verwendung eines 16-fach mengenassoziativen Caches mit unterschiedlichen Zeilengrößen. Im Vergleich zu Implementierung (D) verhalten sich die Pipelining-Varianten für kleine Cachegrößen, die es noch nicht ermöglichen, daß die Arbeitsmenge eines Pipelining-Schrittes im Cache gespeichert werden kann, sehr ähnlich wie Implementierung (D). Bei Verwendung kleiner Zeilengrößen kann Implementierung (PipeDE) in diesem Bereich oft eine etwas geringere Cache-Fehlzugriffszahl erreichen als Implementierung (D). Die elementweise Implementierung (PipeDel) führt in diesem Bereich dagegen zu einer höheren Zahl von Cache-Fehlzugriffen, was durch eine Erhöhung der Zeilengröße verstärkt wird. Ab einer bestimmten Cachegröße, bei der genügend Cachezeilen zur Verfügung stehen, um

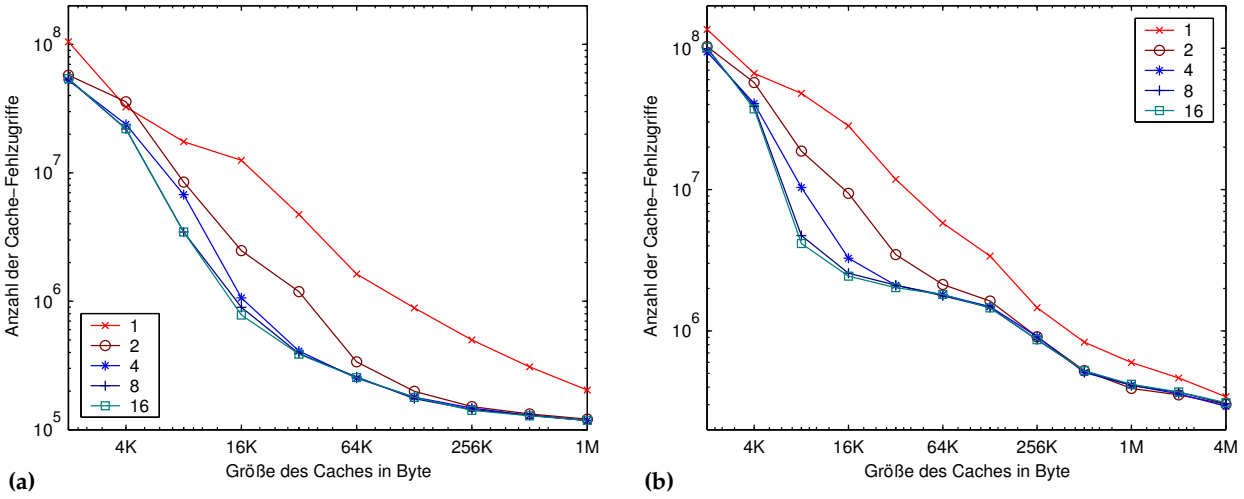


Abb. 4.3: Auswirkung der Assoziativität auf die Anzahl der Cache-Fehlzugriffe für Implementierung (PipeD) bei einer Zeilengröße von 128 Byte. **(a)** MEDAKZO, $N = 15\,000$, **(b)** BRUSS2D-MIX, $N = 250$.

alle Verfahrenskoeffizienten sowie benötigte Stackdaten und Daten von Argumentvektoren zu speichern, die für eine effiziente Ausführung der innersten Schleife erforderlich sind, erreicht Implementierung (D) die geringste Fehlzugriffszahl, bis die Cachegröße auseichend hoch ist, um einen großen Teil der Arbeitsmenge eines Pipelining-Schrittes aufzunehmen. Letzteres tritt abhängig von der Zeilengröße ab einer Cachegröße zwischen 16 KB und 64 KB für MEDAKZO und DOPRI8(7) bzw. zwischen 256 KB und 512 KB für BRUSS2D-MIX und DOPRI5(4) ein. Dies entspricht nahezu exakt der Vorhersage aus der theoretischen Analyse. Nach (4.2) hat der Arbeitsraum eines Pipelining-Schrittes der auf (D) bzw. (Dblock) basierenden Implementierungen im Fall von MEDAKZO und DOPRI8(7) eine Größe von 8,3 KB und im Fall BRUSS2D-MIX und DOPRI5(4) eine Größe von 199,2 KB. Implementierung (PipeDE) besitzt einen nur geringfügig kleineren Arbeitsraum als die übrigen Pipelining-Implementierungen. Das Zugriffsmuster der Implementierung (PipeDE) führt jedoch dazu, daß die Anzahl der erzeugten Cache-Fehlzugriffe höher ist als für die anderen Pipelining-Implementierungen.

Können die während eines Pipelining-Schrittes benutzten Daten im Cache gespeichert werden, erzeugen alle Pipelining-Implementierungen erheblich weniger Cache-Fehlzugriffe als Implementierung (D). Mit Ausnahme der Implementierung (PipeDE) ist ab dieser Cachegröße die Anzahl der Fehlzugriffe für alle untersuchten Pipelining-Implementierungen nahezu gleich. Nur (PipeDE) erzeugt eine etwas größere Anzahl von Fehlzugriffen. Wahrscheinliche Ursache dafür ist, daß (PipeDE) innerhalb eines Pipelining-Schrittes zwar insgesamt weniger Vektorelemente benutzt, die Anzahl der verwendeten Vektoren ist jedoch höher, wodurch eine höhere Anzahl von Konfliktfehlzugriffen provoziert wird.

4.2.6 Vergleich der Laufzeit

Sowohl die theoretische als auch die simulationsbasierte Analyse des Lokalitätsverhaltens weisen auf ein wesentlich verbessertes Lokalitätsverhalten der Pipelining-Implementierungen gegenüber den zuvor betrachteten allgemeinen Implementierungen hin. Um festzustellen, ob dies auch in der Praxis zu einer besseren Laufzeit führt, wurden Laufzeitexperimente auf verschiedenen Rechnersystemen mit unterschiedlicher Architektur durchgeführt.

Auswirkung der Arbeitsmengen auf die Laufzeit

Zunächst betrachten wir die bezüglich der Anzahl der Zeitschritte und der Systemgröße normierte Laufzeit auf einem Itanium-2-SMP vom Typ HP Integrity rx5670 (Beschreibung siehe Anhang A), um die Auswirkung der Arbeitsmengen auf die Laufzeit zu untersuchen. Abbildung 4.7 zeigt die gemessenen normierten Laufzeiten für Implementierung (PipeD) im Vergleich zu den allgemeinen Implementierungen (D)

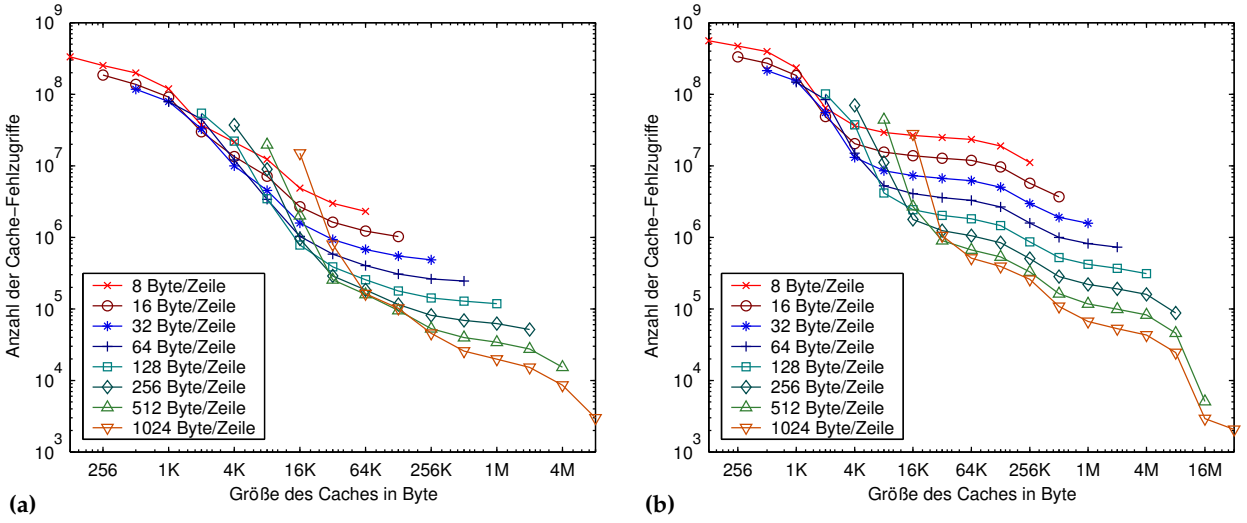


Abb. 4.4: Auswirkung der Zeilengröße auf die Anzahl der Cache-Fehlzugriffe für Implementierung (PipeD) bei Verwendung eines 16-fach mengenassoziativen Caches. **(a)** MEDAKZO, $N = 15\,000$, **(b)** BRUSS2D-MIX, $N = 250$.

und (Dblock) für das Testproblem BRUSS2D-MIX mit dem Verfahren DOPRI5(4). Als weiteres Testproblem wurde MEDAKZO mit dem Verfahren DOPRI8(7) untersucht. Die zugehörigen Meßergebnisse zeigt Abb. 4.8. Wie bereits in den in Abschnitt 2.4 präsentierten Experimenten sind auch hier für Implementierung (Dblock) die Laufzeiten bei Verwendung einer Blockgröße von 5120 dargestellt. Als Blockgröße für die spezialisierten Implementierungen verwenden wir $B = 2N$ für BRUSS2D und $B = 2$ für MEDAKZO.

Für kleine Systemgrößen bis zu ca. 80 000 für BRUSS2D-MIX bzw. ca. 50 000 für MEDAKZO erreichen (D) und (PipeD) sehr ähnliche Laufzeiten, wobei (D) jedoch geringfügig schneller ist. Werden diese Systemgrößen überschritten, können keine 10 vollständigen Vektoren mehr im L3-Cache gespeichert werden. Die normierten Laufzeiten der Implementierung (D) und auch die der Implementierung (Dblock) steigen daher bei weiterer Steigerung der Systemgröße stark an, bis eine Systemgröße von ca. 260 000 erreicht ist, ab der keine 3 vollständigen Vektoren mehr im Cache gespeichert werden können. Zwar steigt auch die normierte Laufzeit der Implementierung (PipeD) in diesem Bereich an, sie erhöht sich für MEDAKZO jedoch, ausgehend von ca. 1,93 µs, nur auf rund 2,0 µs, d. h., sie erhöht sich um nur 3,6 %. Die Laufzeit von (D) erhöht sich dagegen von ca. 1,93 µs auf 2,69 µs, also um 39,4 %. Für Implementierung (Dblock) steigt die Laufzeit ähnlich stark, d. h. um 32,5 % von 1,69 µs auf 2,24 µs. Für BRUSS2D-MIX zeigen sich ähnliche Verhältnisse. Die normierte Laufzeit von (PipeD) steigt hier um ca. 6,8 % von rund 730 ns auf rund 780 ns. Dagegen steigt die normierte Laufzeit der Implementierung (D) von rund 730 ns auf rund 890 ns, d. h. um ca. 22,0 %. Für Implementierung (Dblock) erhöht sich die Laufzeit um etwa 25,8 % von ca. 660 ns auf ca. 830 ns. (PipeD) kann deshalb aufgrund des kleineren Arbeitsraumes der äußeren Schleife für hohe Systemgrößen die geringste Laufzeit erreichen.

Laufzeit auf verschiedenen Zielsystemen

Tabelle 4.3 zeigt die Laufzeiten der Pipelining-Implementierungen (PipeD), (PipeDE) und (PipeDblock) im Vergleich zu den Laufzeiten der allgemeinen Implementierungen (D) und (Dblock) für die Testprobleme MEDAKZO mit dem Verfahren DOPRI8(7) und BRUSS2D mit dem Verfahren DOPRI5(4) für mehrere Problemgrößen auf drei verschiedenen Rechnersystemen, die bereits für den Vergleich der allgemeinen Implementierungen herangezogen wurden (vgl. Abschnitt 2.4.4, Tab. 2.5). Auch hier verwenden wir als Blockgröße für die spezialisierten Implementierungen $B = 2N$ für BRUSS2D. Für MEDAKZO wurde eine Blockgröße von $B = 8$ bei Verwendung einer Problemgröße von $N = 15\,000$ bzw. $B = 16$ bei Verwendung einer Problemgröße von $N = 30\,000$ verwendet.

Testproblem	N	(D)	(Dblock)	(PipeD)	(PipeDE)	(PipeDblock)
Sun Blade 1000, Sun UltraSPARC III Cu, 1050 MHz						
BRUSS2D-MIX	250	28,2	21,5	25,7	25,7	20,4
BRUSS2D-ROW	250	31,7	25,0	—	—	—
BRUSS2D-MIX	500	602,1	379,3	417,4	411,6	319,4
BRUSS2D-ROW	500	653,6	439,0	—	—	—
BRUSS2D-MIX	750	3059,7	1918,0	2106,8	2107,8	1619,7
BRUSS2D-ROW	750	3443,4	2301,5	—	—	—
BRUSS2D-MIX	1000	10018,8	6160,3	6839,2	6698,0	5072,6
BRUSS2D-ROW	1000	11133,8	7310,7	—	—	—
MEDAKZO	15000	116,8	95,3	109,8	111,8	103,1
MEDAKZO	30000	981,0	736,6	871,2	866,7	815,4
IBM eServer pSeries 690, IBM Power 4+, 1,7 GHz						
BRUSS2D-MIX	250	7,3	7,3	7,0	7,1	6,1
BRUSS2D-ROW	250	8,7	8,4	—	—	—
BRUSS2D-MIX	500	117,1	114,4	111,2	111,5	103,5
BRUSS2D-ROW	500	136,4	127,7	—	—	—
BRUSS2D-MIX	750	631,4	662,0	568,2	578,2	530,9
BRUSS2D-ROW	750	747,2	743,9	—	—	—
BRUSS2D-MIX	1000	1929,2	2265,0	1809,6	1829,9	1701,2
BRUSS2D-ROW	1000	2242,5	2679,0	—	—	—
MEDAKZO	15000	56,2	57,9	53,0	52,7	49,5
MEDAKZO	30000	441,0	438,7	403,2	398,3	367,4
MEGWARE Saxonid C2, AMD Opteron, 2 GHz						
BRUSS2D-MIX	250	8,7	8,7	8,2	8,0	8,0
BRUSS2D-ROW	250	7,9	9,1	—	—	—
BRUSS2D-MIX	500	133,4	133,5	124,5	122,8	121,9
BRUSS2D-ROW	500	117,2	134,6	—	—	—
BRUSS2D-MIX	750	666,3	672,3	629,8	615,2	619,5
BRUSS2D-ROW	750	590,3	678,3	—	—	—
BRUSS2D-MIX	1000	2124,2	2125,3	1984,8	1981,6	1988,7
BRUSS2D-ROW	1000	1979,0	2275,2	—	—	—
MEDAKZO	15000	50,6	48,9	45,0	45,5	38,5
MEDAKZO	30000	440,8	426,5	388,4	382,1	351,9

Tab. 4.3: Ausführungszeit der sequentiellen Pipelining-Implementierungen im Vergleich zu den Implementierungen (D) und (Dblock) auf verschiedenen Rechnersystemen.

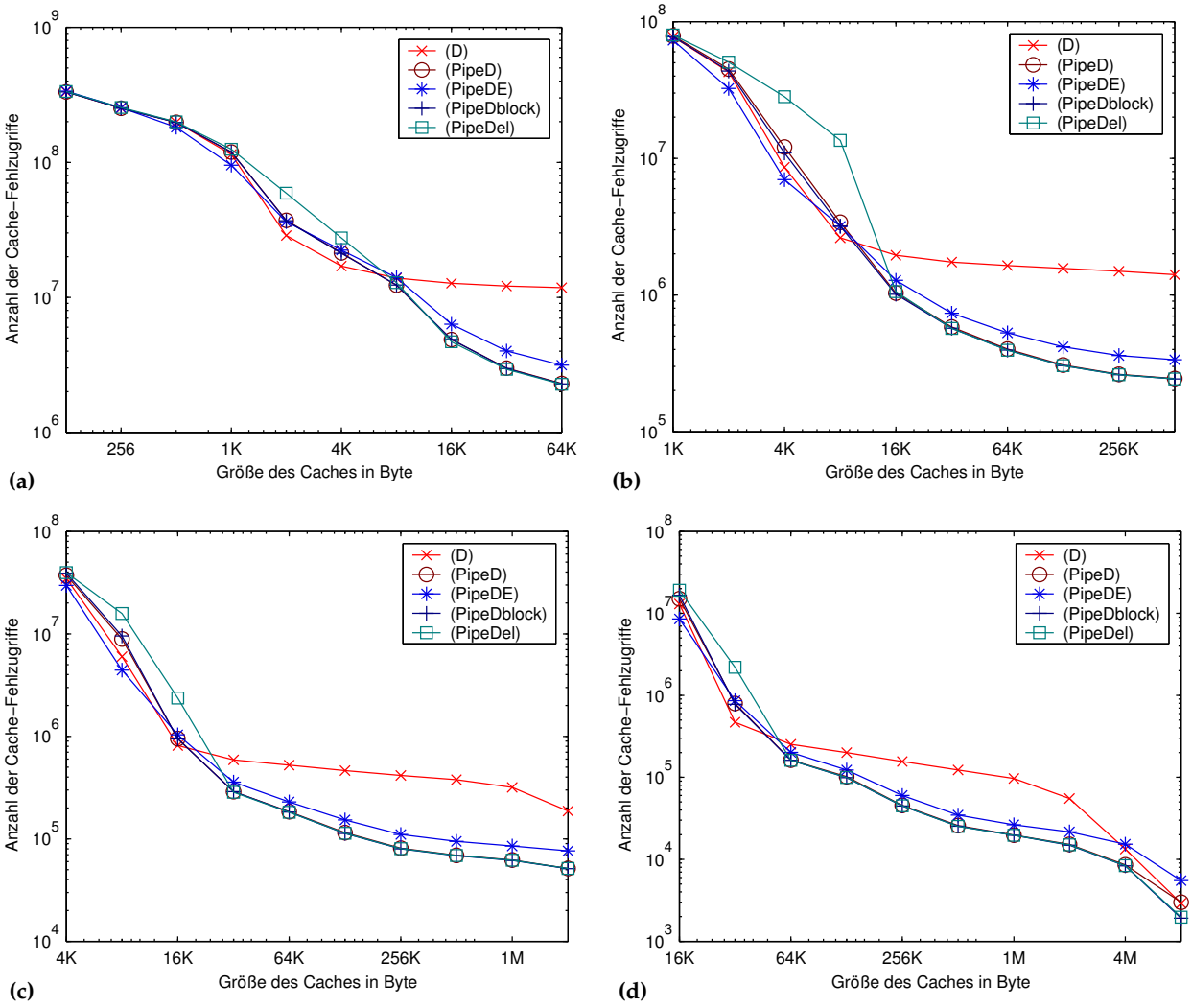


Abb. 4.5: Vergleich der Implementierungen bezüglich der Anzahl der Cache-Fehlzugriffe für MEDAKZO, $N = 15\,000$ bei Verwendung eines 16-fach mengenassoziativen Caches. **(a)** Zeilengröße 8 Byte, **(b)** Zeilengröße 64 Byte, **(c)** Zeilengröße 256 Byte, **(d)** Zeilengröße 1024 Byte.

Sun UltraSPARC III Cu. Bereits auf dem ersten betrachteten Zielsystem, einer Sun Blade 1000 mit zwei UltraSPARC III Cu Prozessoren, erreichen alle Pipelining-Implementierungen in allen durchgeführten Experimenten eine erheblich bessere Laufzeit als Implementierung (D). Implementierung (Dblock) ist jedoch ähnlich erfolgreich. Für BRUSS2D-MIX erreicht (Dblock) bessere Laufzeiten als (PipeD) und (PipeDE). Nur die Pipelining-Implementierung (PipeDblock), die wie (Dblock) ein Schleifen-Tiling realisiert, kann noch geringere Laufzeiten erzielen. Für MEDAKZO übertrifft (Dblock) auch (PipeDblock), da hier das Pipelining-Berechnungsschema keine wesentlichen Vorteile bietet. Aufgrund der geringeren Systemgrößen können für MEDAKZO alle Vektoren im L2-Cache gespeichert werden, so daß auch die allgemeinen Implementierungen den L2-Cache effizient nutzen können. Im Vergleich der Pipelining-Implementierung erreicht in allen Experimenten auf diesem System (PipeDblock) die beste Laufzeit. (PipeDE) ist in einigen Experimenten meßbar schneller als (PipeD), jedoch kann auch (PipeD) in einigen Experimenten bessere Laufzeiten als (PipeDE) erreichen.

IBM Power4+. Auf dem zweiten System, einem Server des Typs IBM p690 auf Basis des Power4+-Prozessors, wird das Potential der Pipelining-Implementierungen noch deutlicher. Hier erreichen alle Pipelining-Implementierungen in allen Experimenten eine geringere Laufzeit als die beiden zum Vergleich herangezo-

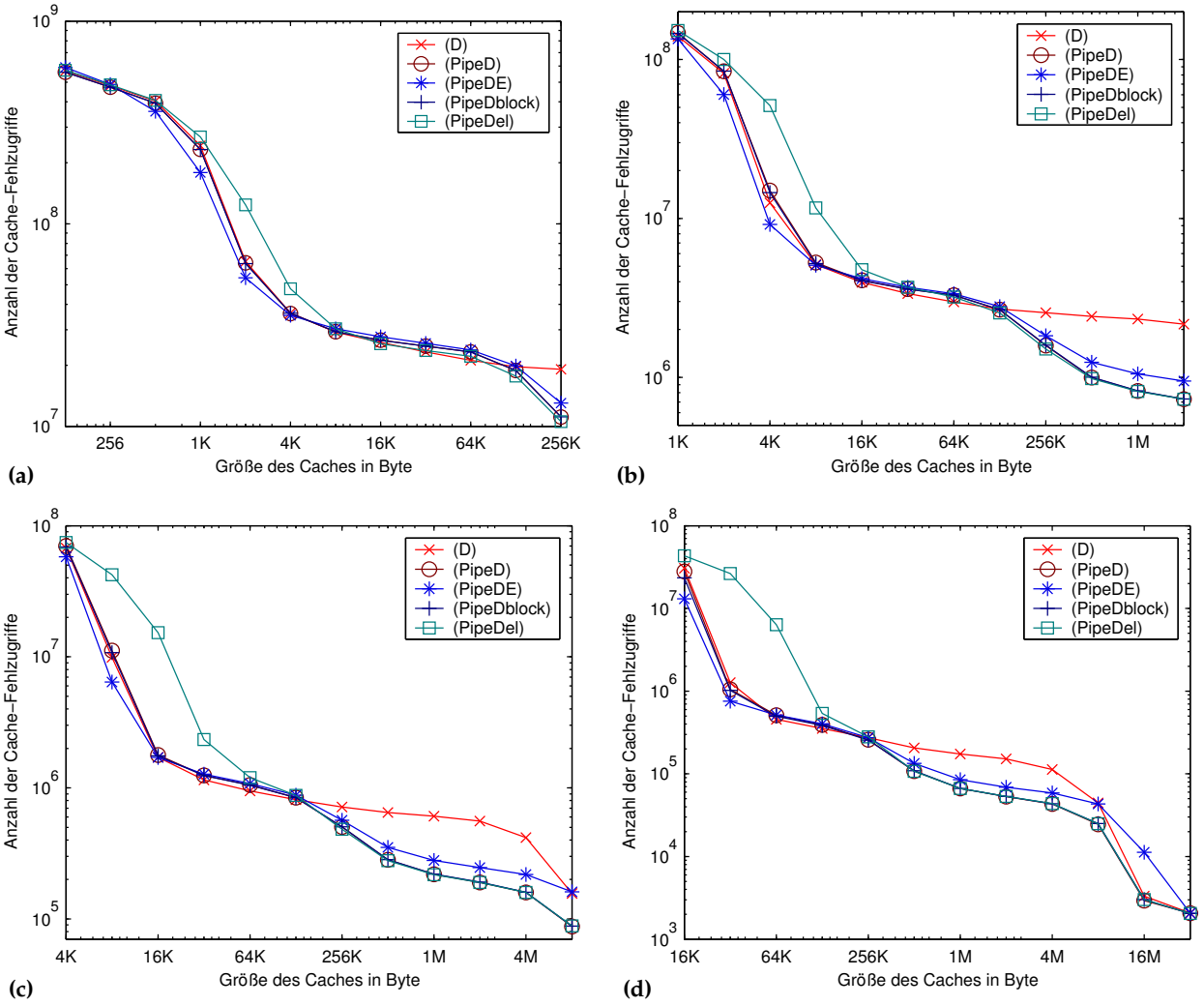


Abb. 4.6: Vergleich der Implementierungen bezüglich der Anzahl der Cache-Fehlzugriffe für BRUSS2D-MIX, $N = 250$ bei Verwendung eines 16-fach mengenassoziativen Caches. **(a)** Zeilengröße 8 Byte, **(b)** Zeilengröße 64 Byte, **(c)** Zeilengröße 256 Byte, **(d)** Zeilengröße 1024 Byte.

genen allgemeinen Implementierungen (D) und (Dblock). Auch auf diesem System erzielt (PipeDblock) in allen Experimenten die geringste Laufzeit aller Pipelining-Implementierungen. (PipeDE) ist in den Experimenten mit BRUSS2D-MIX etwas langsamer als (PipeD), in den Experimenten mit MEDAKZO dagegen geringfügig schneller.

AMD Opteron. Für das letzte betrachtete System, ein mit zwei AMD-Opteron-Prozessoren ausgestattetes Zwei-Wege-SMP-System, ergibt sich ein ähnliches Bild, wie auf den beiden zuvor betrachteten Rechnersystemen. Betrachtet man ausschließlich die beiden Testprobleme BRUSS2D-MIX und MEDAKZO, so sind auch hier in allen Experimenten alle Pipelining-Implementierungen schneller als (D) und (Dblock). Bezieht man jedoch BRUSS2D-ROW in die Betrachtungen mit ein, so kann das Problem BRUSS2D am schnellsten integriert werden, indem man Implementierung (D) zusammen mit BRUSS2D-ROW verwendet. Implementierung (Dblock) ist bei Verwendung von BRUSS2D-ROW dagegen langsamer als bei Verwendung von BRUSS2D-MIX. Im Vergleich der Pipelining-Implementierungen erreicht (PipeDblock) nur für kleinere Problemgrößen die beste Laufzeit. Dies ist in beiden Experimenten mit MEDAKZO und in den Experimenten mit BRUSS2D-MIX für $N = 250$ und $N = 500$ der Fall. Für BRUSS2D-MIX und $N = 750$ sowie $N = 1000$ erreicht die Implementierung (PipeDE) die geringste Laufzeit. In allen Experimenten mit BRUSS2D-MIX

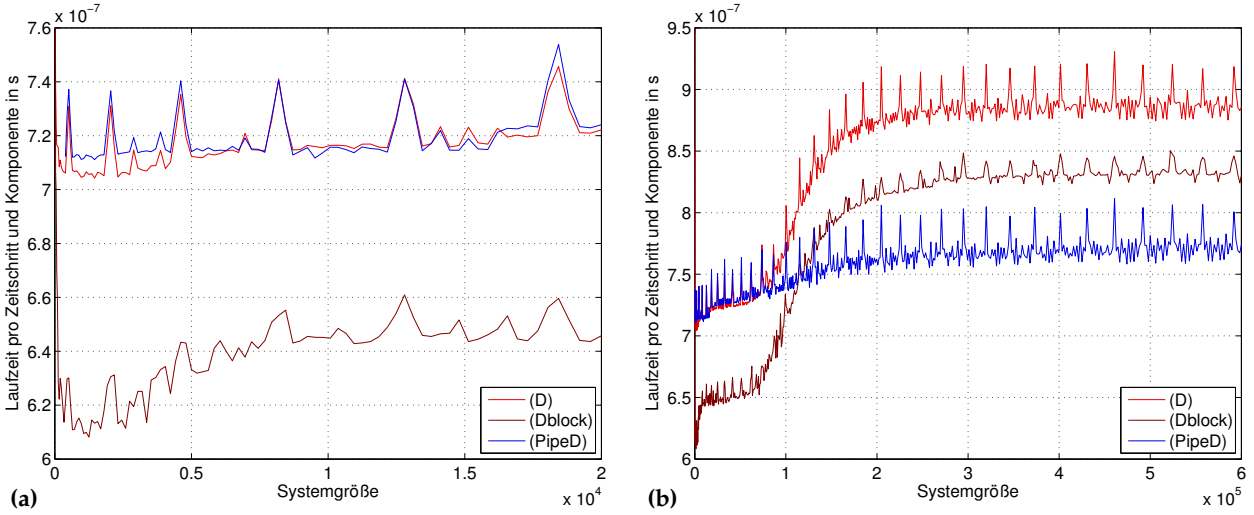


Abb. 4.7: Vergleich der Laufzeit der sequentiellen Implementierungen (D), (Dblock) und (PipeD) pro Zeitschritt und Komponente in Abhängigkeit von der Systemgröße auf einem Itanium-2-System für das Testproblem BRUSS2D-MIX und das Verfahren DOPRI5(4). Die Unterabbildungen zeigen jeweils Ausschnitte der Laufzeitkurve für unterschiedliche Bereiche der Systemgröße.

arbeitet (PipeDE) schneller als (PipeD). Dies trifft auch für MEDAKZO und $N = 30\,000$ zu. Für MEDAKZO und $N = 15\,000$ ist dagegen (PipeD) geringfügig schneller als (PipeDE). In den Experimenten mit BRUSS2D-MIX für $N = 750$ und $N = 1000$, in denen (PipeDblock) nicht die beste Laufzeit erreicht, schneidet (PipeDblock) im Vergleich zu (PipeD) unterschiedlich gut ab. Für $N = 750$ ist (PipeDblock) schneller als (PipeD), für $N = 1000$ ist sie dagegen langsamer.

4.3 Modifikation der Schrittweitenkontrolle

Neben der Verbesserung des Lokalitätsverhaltens bietet die Ausnutzung einer beschränkten Zugriffsdistanz auch das Potential zur Beschleunigung der Ausführung durch die Einsparung von Berechnungen, die im Zusammenhang mit der Schrittweitenkontrolle stehen.

Wenn wir den Pipelining-Algorithmus (Abschnitt 4.2.1) mit besonderem Blick auf die Schrittweitenkontrolle erneut betrachten, stellen wir fest, daß der Pipelining-Algorithmus in jedem Pipelining-Schritt eine neue Komponente bzw. einen Block von Komponenten des Fehlervektors **e** und des Skalierungsvektors **s** berechnet, die beide für die Durchführung der Schrittweitenkontrolle verwendet werden. Innerhalb der Schrittweitenkontrolle wird der Wert

$$\epsilon = \frac{1}{\mathrm{TOL}} \max_{j=1,\ldots,n} \left| \frac{\mathbf{e}[j]}{\mathbf{s}[j]} \right|$$

benutzt, um zu entscheiden, ob der aktuelle Zeitschritt verworfen oder akzeptiert wird, und er wird zur Festlegung der neuen Schrittweite für den nächsten durchzuführenden Zeitschritt verwendet.

Die Tatsache, daß in jedem Pipelining-Schritt eine Komponente bzw. ein Block von Komponenten der Vektoren **e** und **s** neu hinzukommt, gibt uns die Möglichkeit, frühzeitig zu erkennen, ob der aktuelle Zeitschritt verworfen werden muß. Dies ist der Fall, sobald für einen Index j der Betrag $|\mathbf{e}[j]/\mathbf{s}[j]|/\mathrm{TOL}$ größer als 1 ist. Tritt dies beispielsweie bereits für die erste Komponente, d. h. für $j = 1$ ein, könnte man die Ausführung des Zeitschrittes bereits nach der Initialisierung der Pipeline abbrechen und alle weiteren Berechnungen des Zeitschrittes einsparen, da zu diesem Zeitpunkt schon bekannt ist, daß der aktuelle Zeitschritt verworfen werden muß. Erkennt man dies jedoch erst nach Berechnung der letzten Komponente von **e** und **s**, ist auf diese Weise keine Einsparung von Berechnungen möglich.

Die Approximation des lokalen Fehlers, ϵ, wird jedoch nicht nur für die Entscheidung, ob der Zeitschritt verworfen oder akzeptiert wird, benötigt. Er ist ebenfalls das Maß für die Festlegung der Schrittweite des nächsten auszuführenden Zeitschrittes. Wenn ein vorzeitiger Abbruch eines Zeitschrittes bei einem Index j

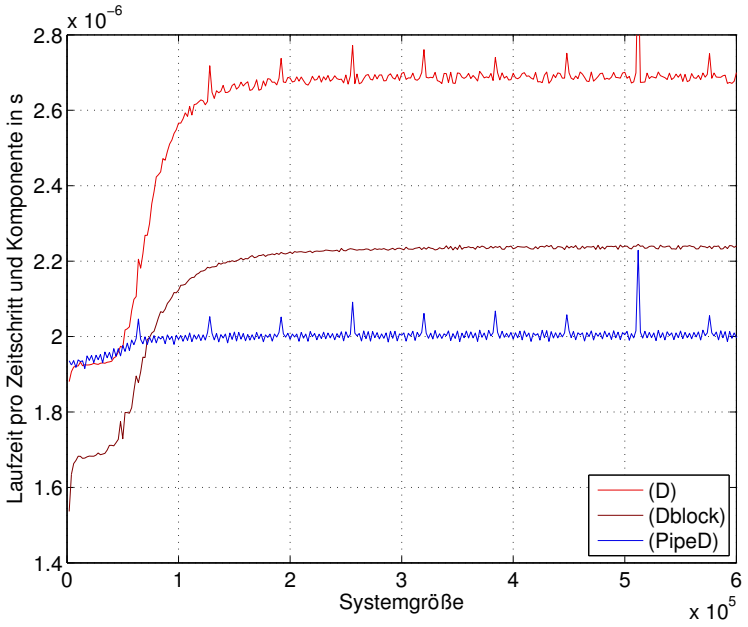


Abb. 4.8: Vergleich der Laufzeit der sequentiellen Implementierungen (D), (Dblock) und (PipeD) pro Zeitschritt und Komponente in Abhängigkeit von der Systemgröße auf einem Itanium-2-System für das Testproblem MEDAKZO mit dem Verfahren DOPRI8(7).

erfolgt, weil erkannt wurde, daß der Zeitschritt verworfen werden muß, ist zum Zeitpunkt des Abbruchs jedoch nur der Wert

$$\epsilon_j = \frac{1}{\mathrm{TOL}} \max_{k=1,\ldots,j} \left| \frac{\mathbf{e}[k]}{\mathbf{s}[k]} \right|$$

bekannt, d. h., die Komponenten $j+1,\ldots,n$ der Vektoren $\mathbf{e}$ und $\mathbf{s}$ wurden noch nicht berücksichtigt. Der endgültige Maximalwert ϵ, der sich bei vollständiger Ausführung des Zeitschrittes ergeben hätte, könnte also größer sein als ϵ_j. Da ϵ in die Berechnung der neuen Schrittweite für die Wiederholung des Zeitschrittes einfließt, kann dies dazu führen, daß die neue Schrittweite zu groß gewählt wird und die Wiederholung des Zeitschrittes ebenfalls verworfen werden muß. In diesem Fall würde der Berechnungsaufwand durch unnötig viele Wiederholungen von Zeitschritten ansteigen, und es könnte kein Laufzeitgewinn erzielt werden. Man könnte versuchen, diesem Problem auszuweichen, indem man ϵ_j nicht in die Berechnung der neuen Schrittweite einbezieht, sondern beispielsweise beim Verwerfen eines Zeitschrittes die alte Schrittweite immer mit einem festen Faktor multipliziert. Dies kann jedoch ebenfalls zu einer ineffizienten Arbeitsweise der Schrittweitenkontrolle führen, da man auf diese Weise den lokalen Fehler nicht berücksichtigt. Es ist daher nicht möglich, die Schrittweite gezielt so zu wählen, daß sie einerseits so groß wie möglich ist, sie andererseits aber auch hinreichend klein ist, damit der maximal zulässige lokale Fehler nicht überschritten wird. Man möchte daher auf eine möglichst genaue Schätzung des lokalen Fehlers nicht verzichten.

Um dennoch Rechenzeit einzusparen, kann man versuchen, mit Hilfe einer Heuristik den zu verwerfenden Zeitschritt erst bei Erreichen eines Index j abzubrechen, ab dem mit hoher Wahrscheinlichkeit

$$\epsilon_j = \epsilon$$

gilt. Dazu kann man die Tatsache ausnutzen, daß die maximale Fehlerkomponente typischerweise eine hohe zeitliche wie auch räumliche Lokalität besitzt, d. h., zwischen aufeinanderfolgenden Zeitschritten wird sich der Index der maximalen Fehlerkomponenten vielfach nur geringfügig verschieben. Weiterhin ist es bei vielen Problemen der Fall, daß die Fehlerkomponenten in der Umgebung der maximalen Fehlerkomponente einen ähnlich hohen Betrag besitzen, der nur wenig kleiner ist als ϵ. Um zu demonstrieren, daß sich dieser Sachverhalt zu einer Einsparung von Berechnungszeit ausnutzen läßt, wurde eine blockweise arbeitende Pipelining-Implementierung mit modifizierter Schrittweitenkontrolle realisiert und untersucht,

welche den Index der maximalen Fehlerkomponente zwischenspeichert. Zu verwerfende Zeitschritte werden frühzeitig abgebrochen, wenn $\epsilon_j > 1$ gilt. Es wird aber immer mindestens so lange weitergerechnet, bis der Block, in dem im vorherigen Zeitschritt der maximale Fehler auftrat, berechnet wurde. Tabelle 4.4 zeigt die Ergebnisse der zugehörigen Laufzeitexperimente, die auf einer Pentium-4-Workstation des Typs Fujitsu-Siemens SCENIC W600 mit dem Testproblem BRUSS2D-MIX durchgeführt wurden.

In den durchgeführten Laufzeitexperimenten konnte durch Einsatz der modifizierten Schrittweitenkontrolle kein signifikanter Laufzeitgewinn erzielt werden. Die Ursache dafür ist, daß ein Laufzeitgewinn nur für verworfene Zeitschritte erzielt werden kann, die verwendete Vorschrift zur Festlegung der Schrittweite jedoch nur zu einer relativ geringen Zahl verworfener Zeitschritte im Vergleich zur Gesamtzahl der Schritte führt. Akzeptierte Zeitschritte werden durch die modifizierte Schrittweitenkontrolle sogar verlangsamt, da zusätzlich zu den bisherigen Berechnungen noch die Entscheidung getroffen werden muß, ob der aktuelle Zeitschritt fortgeführt oder abgebrochen werden soll. Dies führt dazu, daß die Gesamtlaufzeit für das 3-stufige Verfahren RKF2(3) (siehe z. B. Strehmel u. Weiner 1995) ansteigt, da hier aufgrund der geringeren Ordnung eine höhere Gesamtzahl von Zeitschritten ausgeführt wird als für die anderen beiden untersuchten Verfahren, die Zahl der verworfenen Zeitschritte dagegen geringer ist und die Ersparnis bei Abbruch eines verworfenen Zeitschrittes aufgrund der geringen Stufenzahl nur gering ausfällt. Im Gegensatz dazu kann sich für die Verfahren DOPRI5(4) und DOPRI8(7) die Laufzeit in der Regel geringfügig verbessern, oder sie bleibt zumindest nahezu konstant. Der Gewinn liegt jedoch in den meisten Fällen unter 1 %.

Die Laufzeitexperimente zeigen jedoch auch, daß die verwendete einfache Heuristik zur Bestimmung des Index der maximalen Fehlerkomponente für das verwendete Testproblem gut funktioniert. Es trat kein signifikantes Ansteigen der Schrittzahl auf. Man könnte dies gezielt für eine Verbesserung der modifizierten Schrittweitenkontrolle ausnutzen. Ein Nachteil der untersuchten Implementierung besteht darin, daß ein zu verwerfender Zeitschritt unter Umständen erst dann abgebrochen werden kann, wenn bereits nahezu alle für diesen Zeitschritt erforderlichen Berechnungen abgeschlossen sind. Dies ließe sich vermeiden, indem man die Pipeline analog Abb. 4.1 (a) in dem Indexbereich aufbaut, in dem der maximale Fehler vermutet wird. Dies kann jedoch zu einem schlechteren Lokalitätsverhalten führen. Geht man davon aus, daß in der Regel bereits kurz nach dem Aufbau der Pipeline entschieden werden kann, ob ein Abbruch des Zeitschrittes erfolgen soll, wären dafür aber verworfene Zeitschritte extrem preiswert. Dies könnte zu einer signifikanten Laufzeitverbesserung führen, insbesondere dann, wenn eine aggressive Schrittweitenkontrolle eingesetzt wird, die versucht, die Schrittweite so groß wie möglich zu wählen. Es könnte sich eventuell sogar lohnen, die Vorschrift zur Festlegung der Schrittweite gezielt in dieser Hinsicht zu verändern, so daß diese, um eine möglichst hohe Schrittweite zu erzielen, ein häufigeres Verwerfen von Zeitschritten in Kauf nimmt.

4.4 Implementierungen für verteilten Adreßraum

Im folgenden Abschnitt sollen Möglichkeiten zur Ausnutzung einer beschränkten Zugriffsdistanz für die Erstellung paralleler Implementierungen für verteilten Speicher betrachtet werden.

4.4.1 Blockweise Realisierung der Kommunikation

Wie bereits bei der sequentiellen Realisierung des blockweisen Pipelining-Algorithmus (Abschnitt 4.2.1) gehen wir von einer Aufteilung der Stufen- bzw. Argumentvektoren in Blöcke der Größe $B \geq d(\mathbf{f})$ aus. Als Ausgangsimplementierung verwenden wir ebenfalls Implementierung (D). Ähnlich wie in den in Kapitel 3 vorgestellten allgemeinen parallelen Implementierung verwenden wir ebenfalls eine blockweise Datenverteilung, allerdings nicht auf der Basis einzelner Komponenten, sondern auf der Basis von Blöcken der Größe B, so daß jedem Prozessor die durch die lokalen Variablen J_{first} und J_{last} bestimmte Menge von Blocknummern $\{J_{\text{first}}, J_{\text{first}} + 1, \ldots, J_{\text{last}} - 1, J_{\text{last}}\}$ zugeordnet wird.

Da wir von einer beschränkten Zugriffsdistanz ausgehen, greift jede Funktionsauswertung $f_J(t, \mathbf{w})$ für einen Block J nur auf die Blöcke $J - 1$, J und $J + 1$ des Argumentvektors $\mathbf{w}$ zu. Diesen Sachverhalt können wir zur Reduzierung des Kommunikationsaufwandes ausnutzen, denn jetzt ist für einen Prozessor Kommunikation nur noch mit den beiden Prozessoren notwendig, deren Datenbereiche an dessen eigenen Datenbereich angrenzen. Diese Kommunikation mit den beiden Nachbarprozessoren kann mittels Einzel-

Genauigkeits-	Vergleichsimplementierung			modifizierte Schrittweitenkontrolle			
vorgabe	Schritte	verworfen	Laufzeit	Schritte	verworfen	Laufzeit	Gewinn
			RKF2(3)				
10^{-3}	693	11	3,07	693	11	3,08	−0,5 %
10^{-4}	1472	1	6,53	1472	1	6,57	−0,6 %
10^{-5}	3177	2	14,11	3177	2	14,20	−0,6 %
10^{-6}	6849	2	30,39	6849	2	30,64	−0,8 %
10^{-7}	14759	2	65,47	14759	2	65,98	−0,8 %
10^{-8}	31801	3	141,15	31801	3	142,05	−0,6 %
10^{-9}	68514	3	304,32	68514	3	306,48	−0,7 %
10^{-10}	147611	3	656,30	147611	3	658,36	−0,3 %
10^{-11}	318021	4	1412,82	318021	4	1421,82	−0,6 %
10^{-12}	685156	4	3077,89	685156	4	3111,30	−1,1 %
			DOPRI5(4)				
10^{-3}	166	12	2,00	164	10	1,97	1,4 %
10^{-4}	241	10	2,90	241	11	2,89	0,4 %
10^{-5}	367	10	4,40	366	9	4,39	0,0 %
10^{-6}	565	7	6,79	565	7	6,79	0,0 %
10^{-7}	883	6	10,69	883	6	10,59	0,9 %
10^{-8}	1388	5	16,80	1388	5	16,70	0,6 %
10^{-9}	2191	5	26,40	2191	5	26,30	0,4 %
10^{-10}	3461	3	41,78	3461	3	41,66	0,3 %
10^{-11}	5477	2	65,92	5477	2	65,78	0,2 %
10^{-12}	8674	3	104,92	8674	3	104,21	0,7 %
			DOPRI8(7)				
10^{-3}	83	6	2,53	83	6	2,51	0,5 %
10^{-4}	106	5	3,23	106	5	3,22	0,3 %
10^{-5}	144	7	4,39	144	7	4,37	0,5 %
10^{-6}	189	8	5,74	189	8	5,74	0,1 %
10^{-7}	253	10	7,68	253	10	7,68	−0,1 %
10^{-8}	337	13	10,26	338	14	10,28	−0,1 %
10^{-9}	446	9	13,56	446	9	13,55	0,1 %
10^{-10}	918	4	27,94	918	4	27,86	0,3 %
10^{-11}	3162	7	95,95	3162	7	95,91	0,0 %
10^{-12}	10200	9	311,16	10200	9	311,16	0,0 %

Tab. 4.4: Ergebnisse von Laufzeitexperimenten mit einer modifizierten Schrittweitenkontrolle. Die Experimente wurden auf einer Pentium-4-Workstation durchgeführt. Als Testproblem wurde BRUSS2D-MIX mit $N = 100$ und $H = 10,0$ verwendet.

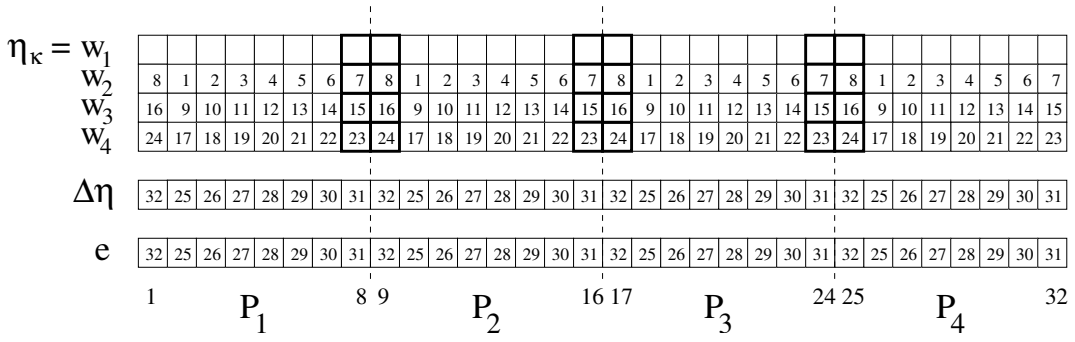


Abb. 4.9: Illustration der Arbeitsweise der parallelen blockweisen Implementierung (Dbc) bei Ausführung auf 4 Prozessoren. Die Numerierung der Blöcke gibt deren Berechnungsreihenfolge an. Blöcke, die von Nachbarprozessoren benötigt werden, sind durch einen breiteren Rand hervorgehoben.

transferoperationen realisiert werden. Dazu müssen in jeder Stufe 2 Blöcke der Größe B versendet und empfangen werden. Die Ausführung von Multibroadcastoperationen ist nicht mehr erforderlich. Dadurch reduzieren sich die Kommunikationskosten von

$$T_p^{\text{comm}}(n) = sT_{\text{allgather}}(p, n/p) + T_{\text{allreduce}}(p, 1) \tag{4.3}$$

auf

$$T_p^{\text{comm}}(n) = 2sT_{\text{send}}(B) + 2sT_{\text{recv}}(B) + T_{\text{allreduce}}(p, 1) \ . \tag{4.4}$$

Sie sind somit nur noch von der Prozessoranzahl p, der Anzahl der Stufen s und der Blockgröße B abhängig, jedoch nicht mehr direkt von der Systemgröße n.

Um die Kommunikationskosten weiter zu reduzieren, können wir versuchen, die Kommunikation parallel zur Durchführung von Berechnungen ablaufen zu lassen, und auf diese Weise die Übertragungszeiten weitestgehend verstecken. Voraussetzung dafür ist, daß die Berechnungsreihenfolge der Argumentvektorblöcke so gewählt wird, daß die Zeitpunkte, an denen zu übertragende Vektorelemente berechnet werden, und die Zeitpunkte, an denen diese von einem anderen Prozessor für weitere Berechnungen benötigt werden, möglichst weit auseinander liegen. Dazu organisieren wir die Berechnungsreihenfolge der Blöcke so, wie es in Abb. 4.9 dargestellt ist, d. h., wir berechnen die Stufen, wie in der Ausgangsimplementierung (D), sequentiell. Wir beginnen die Berechnung einer Stufe jedoch nicht mit dem ersten Block J_{first}, da für die dafür notwendigen Funktionsauswertungen der Block $J_{\text{first}} - 1$ des Argumentvektors der vorhergehenden Stufe benötigt wird, der zunächst noch nicht bekannt ist, weil er dem Vorgängerprozessor zugeordnet ist. Der Block J_{last} kann ebenfalls noch nicht berechnet werden, da dafür der Block $J_{\text{last}} + 1$ des Argumentvektors der vorhergehenden Stufe benötigt wird, welcher dem Nachfolgerprozessor zugeordnet ist. Wir berechnen deshalb zunächst die Blöcke $J_{\text{first}} + 1, \ldots, J_{\text{last}} - 1$, senden aber zuvor mit Hilfe einer nichtblockierenden Sendeoperation die Argumentvektorblöcke J_{first} und J_{last} der vorhergehenden Stufe an den Vorgänger- bzw. den Nachfolgerprozessor. Nach der Berechnung der inneren Blöcke $J_{\text{first}} + 1, \ldots, J_{\text{last}} - 1$ empfangen wir dann die Blöcke $J_{\text{first}} - 1$ und $J_{\text{last}} + 1$ der vorhergehenden Stufe vom Vorgänger bzw. dem Nachfolger und können anschließend die Blöcke J_{first} und J_{last} der aktuellen Stufe berechnen. Den Pseudocode der resultierenden Implementierung, die wir als (Dbc) bezeichnen wollen, ist in den Abb. 4.10 und 4.11 dargestellt.

Die Motivation für dieses Vorgehen besteht darin, daß die Übertragung der Blöcke J_{first} und J_{last} parallel zur Berechnung der inneren Blöcke $J_{\text{first}} + 1, \ldots, J_{\text{last}} - 1$ durchgeführt werden kann und – ein hinreichend schnelles Verbindungsnetzwerk vorausgesetzt – die Übertragungszeit deshalb nicht in die Laufzeit eingeht. Unter dieser Voraussetzung würde sich der Kommunikationsaufwand auf

$$T_p^{\text{comm}}(n) = 2sT_{\text{send}}^{\text{startup}} + 2sT_{\text{recv}}^{\text{startup}} + T_{\text{allreduce}}(p, 1) \tag{4.5}$$

verringern. Als parallele Laufzeit einschließlich des Berechnungsaufwandes erhielten wir somit unter der Annahme $T_{\text{allreduce}}(p, 1) = 1 \cdot T_{\text{allreduce}}(p)$:

$$T_p(n) = T_p^{\text{comp}}(n) + T_p^{\text{comm}}(n) = s\frac{n}{p}T_f(n) + 2sT_{\text{send}}^{\text{startup}} + 2sT_{\text{recv}}^{\text{startup}} + T_{\text{allreduce}}(p) \ . \tag{4.6}$$

```
1:  INIT_STEP();

2:  // Führe die Funktionsauswertungen für die inneren Blöcke der ersten Stufe aus

3:  for (j = J_first + 1; j ≤ J_last − 1; j++) STAGE1(j);

4:  // Führe die Funktionsauswertungen für den ersten Block der ersten Stufe aus
5:  // und stelle das Ergebnis Prozessor me − 1 zur Verfügung

6:  WAIT_FOR_PRED(1);
7:  STAGE1(J_first);
8:  FIRST_BLOCK_COMPLETE(2);

9:  // Führe die Funktionsauswertungen für den letzten Block der ersten Stufe aus
10: // und stelle das Ergebnis Prozessor me + 1 zur Verfügung

11: WAIT_FOR_SUCC(1);
12: STAGE1(J_last);
13: LAST_BLOCK_COMPLETE(2);

14: for (i = 2; i ≤ s; i++)
15: {
16:     // Führe die Funktionsauswertungen für die inneren Blöcke der Stufe i aus

17:     for (j = J_first + 1; j ≤ J_last − 1; j++) STAGE(j, i);

18:     // Führe die Funktionsauswertungen für den ersten Block der Stufe i aus
19:     // und stelle das Ergebnis Prozessor me − 1 zur Verfügung

20:     WAIT_FOR_PRED(i);
21:     STAGE(J_first, i);
22:     FIRST_BLOCK_COMPLETE(i + 1);

23:     // Führe die Funktionsauswertungen für den letzten Block der Stufe i aus
24:     // und stelle das Ergebnis Prozessor me + 1 zur Verfügung

25:     WAIT_FOR_SUCC(i);
26:     STAGE(J_last, i);
27:     LAST_BLOCK_COMPLETE(i + 1);
28: }
```

Abb. 4.10: Parallele Implementierung (Dbc). Die Definition der Makros STAGE() und STAGE1() entspricht der in der sequentiellen Implementierung (PipeD) verwendeten Definition (siehe Abb. 4.2). Die Definition der zur Durchführung der Kommunikation verwendeten Makros ist in Abb. 4.11 dargestellt.

```
1: #define INIT_STEP()                          // Sende verfügbare Blöcke von w_1 an Nachbarprozessoren
2: {
3:     nonblocking_receive(Block J_first − 1 von w_1, me − 1);
4:     nonblocking_receive(Block J_last + 1 von w_1, me + 1);

5:     nonblocking_send(Block J_first von w_1, me − 1);
6:     nonblocking_send(Block J_last von w_1, me + 1);
7: }

8: #define WAIT_FOR_PRED(m)  // Warte, bis der letzte Block von w_m des Vorgängers empfangen wurde
9: {
10:    wait_receive(Block J_first − 1 von w_m, me − 1);
11: }

12: #define WAIT_FOR_SUCC(m)  // Warte, bis der erste Block von w_m des Nachfolgers empfangen wurde
13: {
14:    wait_receive(Block J_last + 1 von w_m, me + 1);
15: }

16: #define FIRST_BLOCK_COMPLETE(m)              // Der erste Block von w_m wurde berechnet
17: {
18:    if (m ≤ s) nonblocking_receive(Block J_first − 1 von w_m, me − 1);

19:    wait_send(Block J_first von w_{m−1}, me − 1);
20:    if (m ≤ s) nonblocking_send(Block J_first von w_m, me − 1);
21: }

22: #define LAST_BLOCK_COMPLETE(m)               // Der letzte Block von w_m wurde berechnet
23: {
24:    if (m ≤ s) nonblocking_receive(Block J_last + 1 von w_m, me + 1);

25:    wait_send(Block J_last von w_{m−1}, me + 1);
26:    if (m ≤ s) nonblocking_send(Block J_last von w_m, me + 1);
27: }
```

Abb. 4.11: Makros zur Realisierung der blockweisen Kommunikation für verteilten Adreßraum.

Weiterhin erhalten wir den Speedup

$$\begin{aligned} S_p(n) = \frac{T_{\text{seq}}(n)}{T_p(n)} &= \frac{snT_f(n)}{s\frac{n}{p}T_f(n) + 2sT_{\text{send}}^{\text{startup}} + 2sT_{\text{recv}}^{\text{startup}} + T_{\text{allreduce}}(p)} \\ &= \left(\frac{1}{p} + \frac{2}{nT_f(n)}\left(T_{\text{send}}^{\text{startup}} + T_{\text{recv}}^{\text{startup}}\right) + \frac{1}{snT_f(n)}T_{\text{allreduce}}(p)\right)^{-1} \end{aligned} \tag{4.7}$$

und die parallele Effizienz

$$E_p(n) = \frac{S_p(n)}{p} = \left(1 + \frac{2p}{nT_f(n)}\left(T_{\text{send}}^{\text{startup}} + T_{\text{recv}}^{\text{startup}}\right) + \frac{p}{snT_f(n)}T_{\text{allreduce}}(p)\right)^{-1}. \tag{4.8}$$

Da nun auch die Blockgröße nicht mehr in die Kommunikationszeit eingeht, ist auch ein indirekter Zusammenhang der Kommunikationszeit mit der Systemgröße ausgeschlossen. Auf die theoretische Skalierbarkeit für eine gegen unendlich strebende Prozessorzahl bei konstanter Systemgröße hat dies allerdings kaum Einfluß, da diese von der Laufzeit der Allreduce-Operation bestimmt wird. In der Praxis kann jedoch die Prozessoranzahl nicht über die Systemgröße hinaus erhöht werden, so daß in einem realistischen Szenario Speedup und Effizienz der allgemeinen Implementierungen immer auch durch die Allgather-Operation bestimmt werden. Die nachfolgend in Abschnitt 4.4.4 dargestellten Laufzeitexperimente zeigen dies, da auf realen Parallelrechnern eine erhebliche Verbesserung der Skalierbarkeit erreicht werden konnte, weil die Allgather-Operation aufgrund der im Vergleich zur Allreduce-Operation großen Menge zu übertragender Daten die Laufzeit wesentlich bestimmt.

Durch die Reduzierung des Kommunikationsaufwandes innerhalb der Stufenberechnung auf einen konstanten Betrag fällt es nun wesentlich leichter, Speedup und Effizienz durch Erhöhung der Problemgröße bei gleichbleibender Prozessoranzahl zu verbessern. Wenn wir auf diese Weise vorgehen, streben Speedup und Effizienz asymptotisch immer gegen die Idealwerte p und 1. Wir sind nicht mehr darauf angewiesen, daß die Funktionsauswertungszeit $T_f(n)$ mit n wächst. Läßt man die Prozessorzahl im gleichen Verhältnis wie die Systemgröße wachsen, d. h. $n/p = c = \Theta(1) \geq 1$, wird die Skalierbarkeit aber nach wie vor durch das Verhältnis von $T_f(n)$ und $T_{\text{allreduce}}(p)$ bestimmt.

4.4.2 Parallele Realisierung des Pipelining-Algorithmus

Da wir zur Realisierung der Implementierung (Dbc) eine Berechnungsreihenfolge gewählt haben, die wie im Fall der allgemeinen Implementierungen die Stufen sequentiell berechnet, besitzt diese auch ein sehr ähnliches Lokalitätsverhalten wie die allgemeinen Implementierungen, d. h. speziell wie die Ausgangsimplementierung (D). Wir betrachten deshalb in diesem Abschnitt eine parallele Realisierung des Pipelining-Algorithmus aus Abschnitt 4.2.1, die ein besseres Lokalitätsverhalten aufweist, jedoch die Kommunikation auf ähnlich effiziente Weise durchführt. Diese Implementierung, die wir wie die sequentielle Implementierungsvariante als (PipeD) bezeichnen wollen, arbeitet wie in Abb. 4.12 veranschaulicht. Den Pseudocode dieser Implementierung zeigen die Abb. 4.13 und 4.11.

Zunächst berechnet jeder Prozessor alle die Blöcke der Argumentvektoren $\mathbf{w}_2, \ldots, \mathbf{w}_s$ und der Vektoren $\Delta\eta$ und $\mathbf{e}$, für deren Berechnung keine Blöcke von Nachbarprozessoren benötigt werden. Dazu wird analog Abb. 4.1 (a) die Pipeline initialisiert und ein diagonaler Lauf über den jeweiligen Teilbereich der Stufen durchgeführt. Damit jeder Prozessor in dieser Phase mindestens einen Block von $\mathbf{w}_s$ berechnet, nehmen wir zu Vereinfachung an, daß jedem Prozessor mindestens ein Datenbereich der Größe $n_B = 2s$ zugeordnet wird. Kommunikation zwischen Nachbarprozessoren ist erst während der Finalisierung der Pipelines notwendig, die an beiden Enden der Pipelines durchgeführt werden muß. Um eine effiziente Kommunikation zu ermöglichen, finalisieren Prozessoren mit gerader Prozessornummer zuerst das Ende ihrer Pipeline mit höherem Index und erst danach das Ende ihrer Pipeline mit niedrigerem Index und Prozessoren mit ungerader Prozessornummer gehen umgekehrt vor. Dadurch arbeiten benachbarte Prozessoren immer an benachbarten Pipeline-Enden und schließen so die nach Abschluß des Sweeps noch vorhandene Lücke nach Art der Arbeitsweise eines Reißverschlusses. Die Berechnungsreihenfolge der Blöcke während der Finalisierung entspricht dabei im wesentlichen der in der sequentiellen Implementierungsvariante verwendeten Berechnungsreihenfolge, wobei die Nachbarprozessoren jedoch spiegelbildlich zueinander arbeiten.

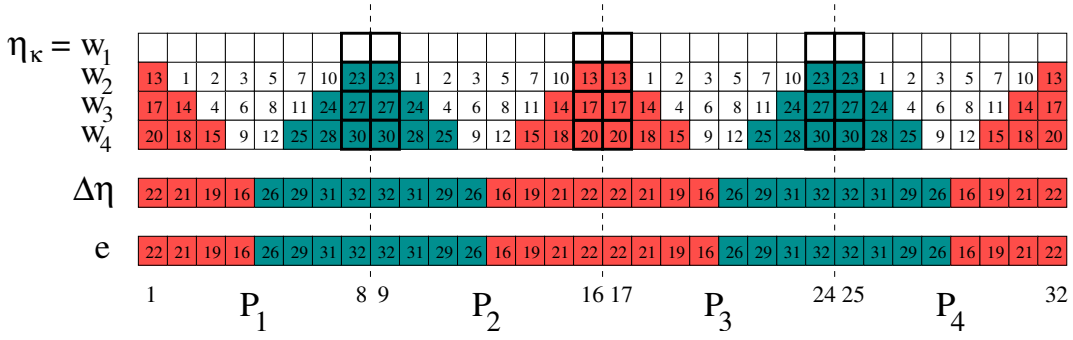


Abb. 4.12: Illustration der Arbeitsweise der parallelen Pipelining-Implementierung (PipeD) bei Ausführung auf 4 Prozessoren. Die Numerierung der Blöcke gibt deren Berechnungsreihenfolge an. Blöcke, auf die während der Finalisierungsphase zugegriffen wird, sind als ausgefüllte Rechtecke dargestellt, und Blöcke, die von Nachbarprozessoren benötigt werden, sind durch einen breiteren Rand hervorgehoben.

Wie in Implementierung (Dbc) werden die Blöcke J_{first} und J_{last} eines Argumentvektors mittels einer nichtblockierenden Sendeoperation an den Vorgänger- bzw. Nachfolgerprozessor abgesandt, sobald sie fertig berechnet wurden. Der Abschluß der Empfangsoperation erfolgt erst dann, wenn diese Blöcke für Berechnungen benötigt werden. Auf diese Weise kann die Übertragung dieser Blöcke parallel zur Berechnung einer Diagonale erfolgen, wobei sich allerdings die Länge der Diagonalen in jedem Finalisierungsschritt um einen Block verkürzt. Werden im ersten Finalisierungsschritt nach dem Absenden von Block J_{first} bzw. J_{last} von $\mathbf{w}_2$ noch $s-1$ Blöcke berechnet, bis dieser im nächsten Finalisierungsschritt vom entsprechenden Nachbarprozessor benötigt wird, ist es im vorletzten Finalisierungsschritt nur noch 1 Block. Hinzu kommt, daß Blöcke einer höheren Stufe einen geringeren Berechnungsaufwand erfordern als Blöcke auf einer niedrigeren Stufe. Die Übertragung der Blöcke muß deshalb wesentlich schneller durchgeführt werden als im Fall der Implementierung (Dbc), um die Übertragungszeiten vollständig durch Berechnungen überdecken zu können.

4.4.3 Alternative Finalisierung der Pipelines

Die parallele Implementierung (PipeD) führt innerhalb des Berechnungskernels $2s$ Kommunikationsoperationen aus, deren Startupzeiten in die Programmlaufzeit einfließen. Da auf einigen Parallelrechnern mit verteiltem Adreßraum die Startupzeiten der Kommunikationsoperationen sehr hoch sind, könnten auf solchen Systemen alternative Implementierungsvarianten, die weniger Kommunikationsoperationen ausführen, eine geringere Laufzeit erreichen, auch wenn sie deshalb eine größere Datenmenge übertragen müssen.

Eine Möglichkeit, dies zu realisieren, besteht darin, die Finalisierung der Pipelines so zu organisieren, daß die beteiligten Prozessoren jeweils das Ende ihrer eigenen Pipeline mit höherem Index und das benachbarte Ende der Pipeline des Nachfolgerprozessors finalisieren. Die dazu benötigten Blöcke des Nachfolgerprozessors werden während der Initialisierung seiner Pipeline berechnet und können im Anschluß an die Initialisierung parallel zur Sweep-Phase übertragen werden. Diese Übertragung umfaßt $2s + \sum_{i=2}^{s}(i-1)$ Argumentvektorblöcke und je s Blöcke von $\Delta\eta$, $\mathbf{e}$ und $\mathbf{s}$, also insgesamt $s(s+9)/2$ Blöcke. Da die zu übertragenden Blöcke nicht zusammenhängend im Speicher abgelegt sind, müssen sie vor dem Versenden in einen zusammenhängenden Puffer kopiert und nach dem Empfangen wieder aus einem Puffer extrahiert werden. Dies führt zu einem zusätzlichen Overhead. Parallel zu dieser Datenübertragung können $s(n_B/p - 2s)$ Blöcke innerhalb der Sweep-Phase berechnet werden. Abbildung 4.14 illustriert an einem Beispiel die Berechnungsreihenfolge und die zu übertragenden Datenblöcke. Aus Platzgründen wurde in der Darstellung die Systemgröße so gewählt, daß jedem Prozessor gerade $2s$ Blöcke zugeordnet werden, wodurch keine Sweep-Phase stattfindet.

Während der Finalisierung berechnen die Prozessoren Blöcke von $\Delta\eta$, die der Nachfolgerprozessor im Falle des Akzeptierens des Zeitschrittes zur Aktualisierung des Vektors η benötigt und die deshalb an den Nachfolgerprozessor gesandt werden müssen. Dies wurde in zwei Implementierungsvarianten auf unterschiedlliche Weise realisiert. Die in Abb. 4.15 dargestellte Implementierung (PipeD2) überträgt diese Daten nur dann, wenn der Zeitschritt akzeptiert wird. Dies hat den Vorteil, daß die Datenübertragung nur in

```
1:  INIT_STEP();

2:  // Initialisierung der Pipeline

3:  for (j = 1; j < s; j++)
4:  {
5:      STAGE1((J_first + 2j − 1));
6:      for (i = 2; i ≤ j; i++)
7:          STAGE(J_first + 2j − i, i);

8:      STAGE1(J_first + 2j);
9:      for (i = 2; i ≤ j; i++)
10:         STAGE(J_first + 2j − i + 1, i);
11: }

12: // Diagonaler Sweep über die Argumentvektoren

13: for (j = J_first + 2s − 1; j ≤ J_last − 1; j++)
14: {
15:     STAGE1(j);
16:     for (i = 2; i ≤ s; i++)
17:         STAGE(j − i + 1, i);
18: }

19: // Finalisierung der Pipeline

20: if (me mod 2 = 0) goto finalize_high;

21: label finalize_low;

22: WAIT_FOR_PRED(1);
23: STAGE1(J_first);
24: FIRSTBLOCK_COMPLETE(2);

25: for (i = 2; i ≤ s; i++) STAGE(J_first + i − 1, i);

26: for (j = 2; j ≤ s; j++)
27: {
28:     WAIT_FOR_PRED(j);
29:     STAGE(J_first, j);
30:     FIRSTBLOCK_COMPLETE(j + 1);

31:     for (i = j + 1; i ≤ s; i++)
32:         STAGE(J_first + i − j, i);
33: }

34: if (me mod 2 = 0) goto error_control;

35: label finalize_high;

36: WAIT_FOR_SUCC(1);
37: STAGE1(J_last);
38: LASTBLOCK_COMPLETE(2);

39: for (i = 2; i ≤ s; i++) STAGE(J_last − i + 1, i);

40: for (j = 2; j ≤ s; j++)
41: {
42:     WAIT_FOR_SUCC(j);
43:     STAGE(J_last, j);
44:     LASTBLOCK_COMPLETE(j + 1);

45:     for (i = j + 1; i ≤ s; i++)
46:         STAGE(J_last − i + j, i);
47: }

48: if (me mod 2 = 0) goto finalize_low;

49: label error_control;
```

Abb. 4.13: Parallele Implementierung (PipeD). Die Definition der Makros STAGE() und STAGE1() entspricht der in der sequentiellen Implementierung (PipeD) verwendeten Definition (siehe Abb. 4.2). Die Definition der zur Durchführung der Kommunikation verwendeten Makros ist in Abb. 4.11 dargestellt.

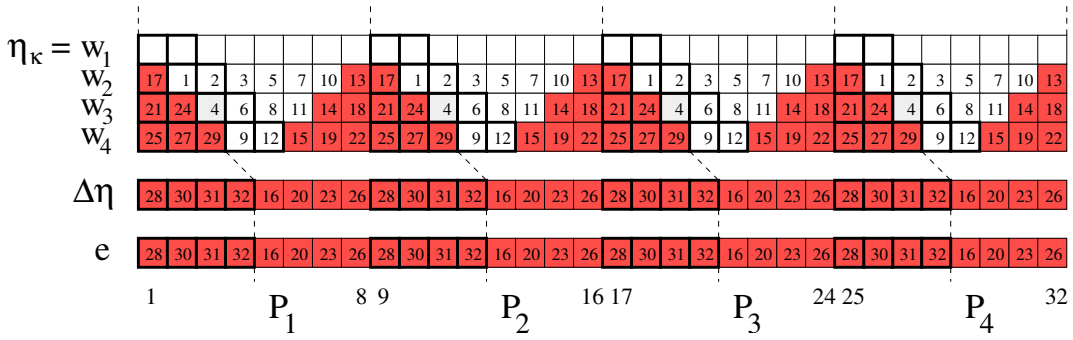


Abb. 4.14: Illustration der Arbeitsweise der parallelen Pipelining-Implementierungen (PipeD2) und (PipeD3) bei Ausführung auf 4 Prozessoren. Die Numerierung der Blöcke gibt deren Berechnungsreihenfolge an. Blöcke, auf die während der Finalisierungsphase zugegriffen wird, sind als ausgefüllte Rechtecke dargestellt, und Blöcke, die von Nachbarprozessoren benötigt werden, sind durch einen breiteren Rand hervorgehoben.

solchen Zeitschritten durchgeführt wird, in denen die übertragenen Daten tatsächlich benötigt werden. Gleichzeitig ergibt sich dadurch aber auch der Nachteil, daß die Datenübertragung erst dann durchgeführt werden kann, wenn die Daten bereits benötigt werden, d. h., die Datenübertragungszeit kann nicht durch Berechnungen überdeckt werden. Demgegenüber führt die in Abb. 4.16 dargestellte Implementierung (PipeD3) die Übertragung der Blöcke von $\Delta\eta$ in jedem Zeitschritt durch, sobald die Berechnung aller benötigten Blöcke abgeschlossen ist. Dadurch wird zwar im Fall des Verwerfens des Zeitschrittes ein unnötiger Datenaustausch durchgeführt, es kann aber zumindest ein Teil der Übertragungszeit durch Berechnungen überdeckt werden.

4.4.4 Laufzeitvergleich auf verschiedenen Zielsystemen

Zur Untersuchung des Laufzeitverhaltens der parallelen Implementierungen für verteilten Adreßraum, die ausgehend von einer beschränkten Zugriffsdistanz eine effizientere Kommunikation realisieren, wurden Laufzeitexperimente auf dem Chemnitzer Linux Cluster (CLiC), einer Cray T3E-1200, einem Opteron-SMP-Cluster, dem Jülicher Multiprozessor (JUMP) sowie einer Sun Fire 6800 durchgeführt. Als Testprobleme werden BRUSS2D-MIX und MEDAKZO verwendet, wobei für BRUSS2D-MIX als Blockgröße B die jeweilige Zugriffsdistanz $d(\mathbf{f}) = 2N$ und für MEDAKZO eine Blockgröße von $B = 16$ benutzt wird. Als Referenz für die Speedup-Berechnung wird die schnellste sequentielle Variante der jeweils betrachteten Implementierungen verwendet. Da dies auch die sequentielle Implementierung (PipeD) einschließt, die häufig eine geringere Laufzeit erzielt als die allgemeinen Implementierungen, zeigen die Abbildungen in diesem Abschnitt für die allgemeinen Implementierungen oft geringere Speedups als in den in Abschnitt 3.3.4 präsentierten Experimenten, in denen nur die allgemeinen Implementierungen untersucht wurden.

Chemnitzer Linux Cluster (CLiC)

Bei der Untersuchung des Laufzeitverhaltens der allgemeinen Implementierung (D) auf dem CLiC (Abschnitt 3.3.4) konnte durch die parallele Ausführung keine Laufzeitverbesserung gegenüber der sequentiellen Ausführung erzielt werden. Die auf eine beschränkte Zugriffsdistanz spezialisierten Implementierungen erreichen demgegenüber eine sehr gute Skalierbarkeit. Abbildung 4.17 zeigt die erreichten Speedups, die unter den gleichen Bedingungen wie in den in Abschnitt 3.3.4 präsentierten Experimenten mit Implementierung (D) gemessen wurden. Als Testproblem wurde in allen Experimenten BRUSS2D-MIX verwendet. Insgesamt wurden drei Experimente mit dem 7-stufigen Lösungsverfahren DOPRI5(4) und den Problemgrößen $N = 896$ und $N = 384$ sowie mit dem 13-stufigen Verfahren DOPRI8(7) und der Problemgröße $N = 896$ durchgeführt. Da alle Pipelining-Implementierungen voraussetzen, daß jedem Prozessor mindesten $2s$ Blöcke zugeordnet werden, wurden diese Implementierungen in den durchgeführten Experimenten nur auf bis zu 64 (DOPRI5(4), $N = 896$), 32 (DOPRI8(7), $N = 896$) bzw. 27 (DOPRI5(4), $N = 384$) Prozessoren ausgeführt.

```
1: // Initialisierung der Pipeline

2: for (j = 1; j < s; j++)
3: {
4:     STAGE1(J_first + 2j − 1);
5:     for (i = 2; i ≤ j; i++) STAGE(J_first + 2j − i, i);
6:     STAGE1(J_first + 2j);
7:     for (i = 2; i ≤ j; i++) STAGE(J_first + 2j − i + 1, i);
8: }

9: // Versende Daten am Vorgänger

10: Verpacke alle von Prozessor me − 1 benötigten Blöcke in einen Puffer;
11: Starte nichtblockierende Sendeoperation für diesen Puffer an Prozessor me − 1;

12: // Diagonaler Sweep über die Argumentvektoren

13: for (j = J_first + 2s − 1; j ≤ J_last − 1; j++)
14: {
15:     STAGE1(j);
16:     for (i = 2; i ≤ s; i++) STAGE(j − i + 1, i);
17: }

18: // Empfange Daten vom Nachfolger

19: Warte, bis die vom Nachfolger versandten Daten in einem Puffer empfangen wurden;
20: Entpacke die im Puffer gespeicherten Daten;
21: Warte, bis die nichtblockierende Sendeoperation zum Vorgänger abgeschlossen ist;

22: // Finalisierung der Pipeline

23: STAGE1(J_last);
24: for (i = 2; i ≤ s; i++) STAGE(J_last − i + 1, i);
25: STAGE1((J_last mod n_B) + 1);
26: for (i = 2; i ≤ s; i++) STAGE(J_last − i + 2, i);
27: for (i = 2; i ≤ s; i++)
28: {
29:     for (j = i; j ≤ s; j++) STAGE(((J_last + 2i − j − 2) mod n_B) + 1, j);
30:     for (j = i; j ≤ s; j++) STAGE(((J_last + 2i − j − 1) mod n_B) + 1, j);
31: }

32: // Schrittweitenkontrolle

33: ⋮
34: if (Schritt wird akzeptiert)
35: {
36:     Starte nichtblockierende Sendeoperation für die vom Nachfolger benötigten Blöcke von Δη;
37:     Empfange die selbst benötigten Blöcke von Δη vom Vorgänger;
38:     Warte auf den Abschluß der nichtblockierenden Sendeoperation;
39:     ⋮
```

Abb. 4.15: Parallele Implementierung (PipeD2) für verteilten Adreßraum. Die Definition der Makros STAGE() und STAGE1() entspricht der in der sequentiellen Implementierung (PipeD) verwendeten Definition (siehe Abb. 4.2).

```
1: // Initialisierung der Pipeline und diagonaler Sweep über die Argumentvektoren

2: Führe die gleichen Berechungen wie in Implementierung (PipeD2) aus;

3: // Finalisierung der Pipeline

4: STAGE1((J_last mod n_B) + 1);
5: for (i = 2; i ≤ s; i++) STAGE(((J_last + i − 1) mod n_B) + 1, i);

6: STAGE1(J_last);
7: for (i = 2; i ≤ s; i++) STAGE(((J_last + i − 2) mod n_B) + 1, i);

8: for (i = 2, k = 2; i ≤ s; i++)
9: {
10:     for (j = i; j ≤ s; j++) STAGE(((J_last − k + j − 1) mod n_B) + 1, j);

11:     k++;
12:     if (k = s)
13:         Starte nichtblockierende Sendeoperation für die vom Nachfolger benötigten Blöcke von Δη;

14:     for (j = i; j ≤ s; j++) STAGE(((J_last − k + j − 1) mod n_B) + 1, j);

15:     k++;
16:     if (k = s)
17:         Starte nichtblockierende Sendeoperation für die vom Nachfolger benötigten Blöcke von Δη;
18: }

19: Empfange die selbst benötigten Blöcke von Δη vom Vorgänger;
20: Warte auf den Abschluß der nichtblockierenden Sendeoperation;
```

Abb. 4.16: Parallele Implementierung (PipeD3) für verteilten Adreßraum. Die Definition der Makros STAGE() und STAGE1() entspricht der in der sequentiellen Implementierung (PipeD) verwendeten Definition (siehe Abb. 4.2).

Im Vergleich zueinander zeigen die Implementierungen (Dbc) und (PipeD) in allen drei Experimenten eine sehr ähnliche Skalierbarkeit. Nur in dem Experiment mit DOPRI5(4) und $N = 384$ erreicht (PipeD) bei gleicher Prozessorzahl einen geringfügig höheren Speedup. Für 27 Prozessoren erreichen (Dbc) und (PipeD) in diesem Experiment einen Speedup von 20,6 bzw. 22,4. Bei Verwendung von DOPRI8(7) und $N = 896$ erreichen sie für 32 Prozessoren Speedup-Werte von 29,4 und 29,0. Für DOPRI5(4) und die gleiche Problemgröße beträgt der Speedup für 64 Prozessoren sogar 53,6 bzw. 52,8. Die Pipelining-Implementierungen (PipeD2) und (PipeD3) sind in allen Experimenten langsamer als (Dbc) und (PipeD), erreichen aber im Vergleich zueinander etwa die gleiche Laufzeit. Bei Verwendung von DOPRI5(4) und $N = 384$ wurden für 27 Prozessoren für die beiden Implementierungen (PipeD2) und (PipeD3) Speedups von 10,3 bzw. 10,7 gemessen. Für DOPRI8(7) und $N = 896$ erreichen die beiden Implementierungen für 32 Prozessoren Speedups von 18,6 bzw. 18,8. Die höchsten Speedup-Werte von 25,17 bzw. 25,55 wurden bei Verwendung von 64 Prozessoren für DOPRI5(4) und $N = 896$ gemessen.

Cray T3E-1200

Auf der Cray T3E-1200 konnte die allgemeine Implementierung (D) ihre beste Skalierbarkeit erzielen (vgl. Abschnitt 3.3.4). Dennoch betrug der höchste für Implementierung (D) gemessene Speedup nur 17,7. Die spezialisierten Implementierungen erreichen in den gleichen Experimenten, die bereits mit Implementierung (D) durchgeführt wurden, wesentlich höhere Speedups (sieh Abb. 4.18). Die Laufzeitunterschiede

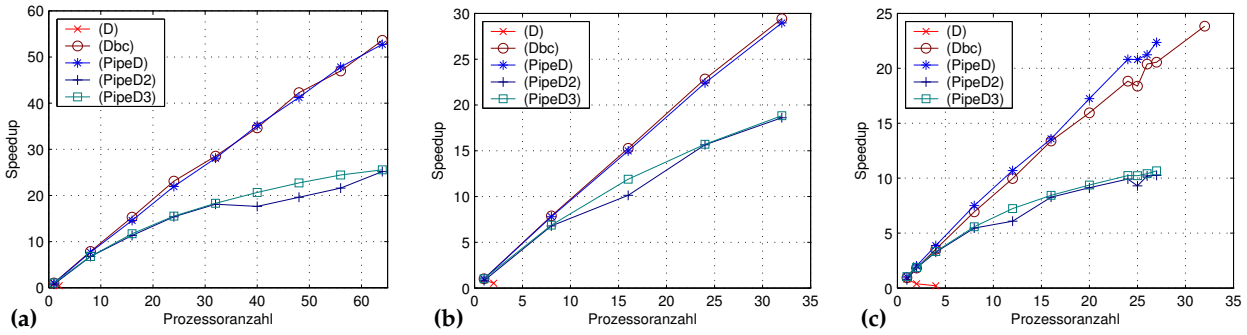


Abb. 4.17: Speedups der parallelen spezialisierten Implementierungen für verteilten Adreßraum im Vergleich zu Implementierung (D) auf dem CLiC. **(a)** DOPRI5(4), $N = 896$, $H = 0,5$, **(b)** DOPRI8(7), $N = 896$, $H = 0,5$, **(c)** DOPRI5(4), $N = 384$, $H = 4,0$.

zwischen den verschiedenen spezialisierten Implementierungen sind in allen durchgeführten Experimenten sehr gering. Für DOPRI5(4) und $N = 384$ erreichen die spezialisierten Implementierungen bei Verwendung von 27 Prozessoren Speedups zwischen 21,6 und 22,1. Für DOPRI8(7) und $N = 896$ können bei Verwendung von 32 Prozessoren Speedups zwischen 28,5 und 28,9 erzielt werden. Wie bereits auf dem CLiC wurden auch auf der Cray T3E-1200 die höchsten Speedup-Werte für DOPRI5(4) und $N = 896$ gemessen, die zwischen 54,2 und 55,4 lagen.

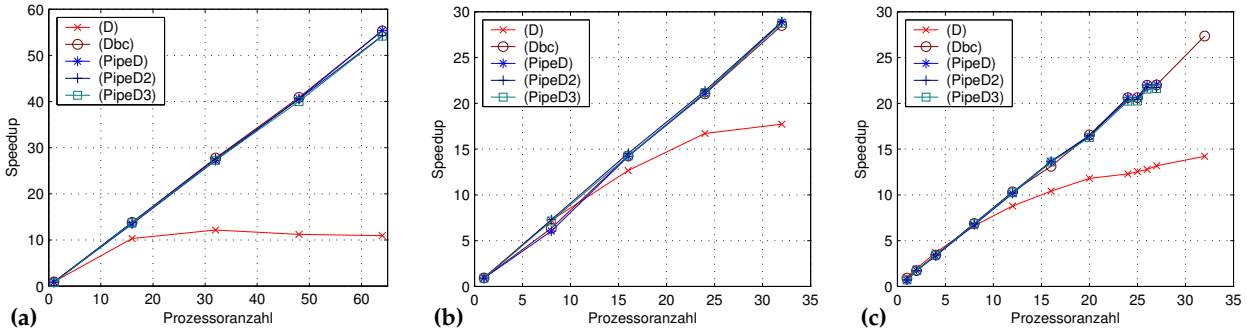


Abb. 4.18: Speedups der parallelen spezialisierten Implementierungen für verteilten Adreßraum im Vergleich zu Implementierung (D) auf einer Cray T3E-1200. **(a)** DOPRI5(4), $N = 896$, $H = 0,5$, **(b)** DOPRI8(7), $N = 896$, $H = 0,5$, **(c)** DOPRI5(4), $N = 384$, $H = 4,0$.

Opteron-SMP-Cluster

Die allgemeinen Implementierungen (A) und (D) zeigten auf diesem System ein ähnlich schlechtes Verhalten wie auf dem CLiC (vgl. Abschnitt 3.3.4). Ein Speedup größer als 1 konnte nur erzielt werden, wenn entweder nur zwei MPI-Prozesse auf ein und demselben Knoten gestartet wurden, oder wenn MPICH 1.2.5 und das Infiniband-Netzwerk zur Ausführung einer größeren Zahl von Prozessen verteilt auf mehrere Knoten benutzt wurden. In allen Fällen waren die für die allgemeinen Implementierungen gemessenen Speedups jedoch kleiner als 2.

Wie die Abb. 4.19 und 4.20 zeigen, erreichen auch auf diesem System die spezialisierten Implementierungen erheblich bessere Speedups als die allgemeinen Implementierungen. Im Experiment mit BRUSS2D-MIX und dem Infiniband-Netzwerk wurden für die beiden MPI-Implementierungen LAM/MPI 7.1.1 und MPICH 1.2.5 nahezu gleiche Speedup-Werte gemessen. Die beste Skalierbarkeit erreichen die Implementierungen (PipeD) und (Dbc), wobei (PipeD) meßbar höhere Speedups von 54,2 bzw. 54,4 erzielen kann, wogegen (Dbc) nur 51,0 bzw. 50,8 erreicht. (PipeD2) und (PipeD3) arbeiten beide etwa gleich schnell, sind im Vergleich zu (PipeD) und (Dbc) jedoch deutlich langsamer. Sie erreichen Speedups zwischen 40,8 und 41,8. Im Experiment mit MEDAKZO und dem Infiniband-Netzwerk sind die Speedups deutlich geringer. Hier erreichen alle spezialisierten Implementierungen unabhängig von der verwendeten MPI-Version ähnlich

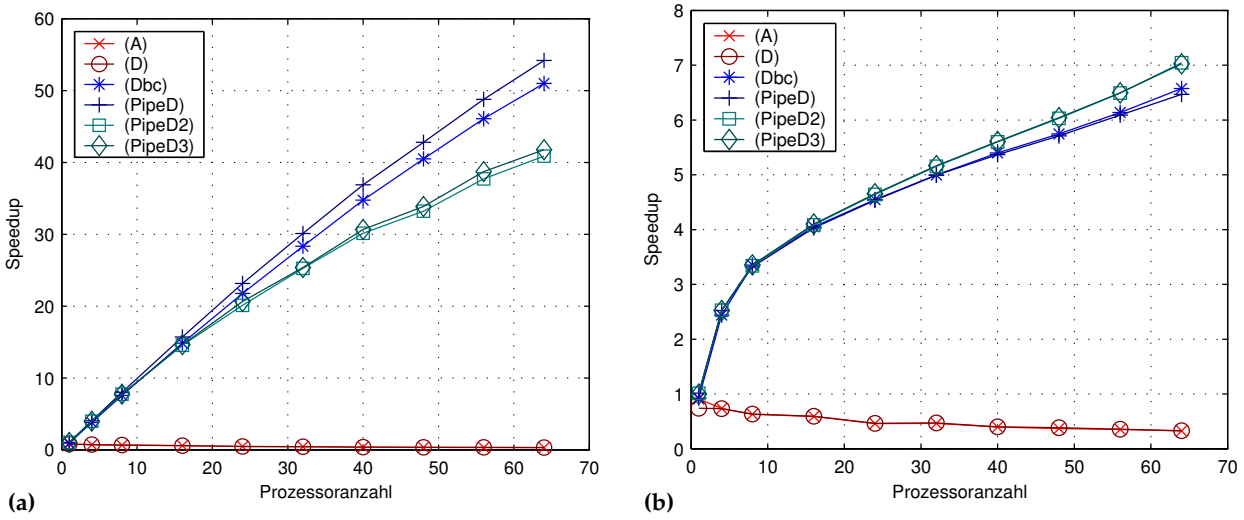


Abb. 4.19: Speedups der parallelen spezialisierten Implementierungen für verteilten Adreßraum im Vergleich zu den Implementierungen (A) und (D) auf einem Opteron-SMP-Cluster bei Verwendung eines Infiniband-Netzwerkes und LAM/MPI 7.1.1. **(a)** BRUSS2D-MIX, DOPRI5(4), $N = 1000$, $H = 4,0$, **(b)** MEDAKZO, DOPRI8(7), $N = 30\,000$, $H = 10^{-4}$.

schlechte Speedup-Werte zwischen 6,5 und 7,1. Diese sind dennoch erheblich besser als die von den allgemeinen Implementierungen erzielten Speedup-Werte. Auffällig ist, daß bei Verwendung von LAM/MPI 7.1.1 die Implementierungen (PipeD2) und (PipeD3) erkennbar schneller sind als (PipeD) und (Dbc), während für MPICH 1.2.5 der umgekehrte Fall zutrifft.

Am Beispiel von LAM/MPI 7.1.1 zeigt Abb. 4.21 für BRUSS2D-MIX und Abb. 4.22 für MEDAKZO, wie sich die Verwendung unterschiedlicher Verbindungsnetzwerke auf die Skalierbarkeit auswirkt. Für BRUSS2D-MIX ist deutlich erkennbar, daß die Geschwindigkeit des Verbindungsnetzwerkes einen erheblichen Einfluß auf die erreichbaren Speedup-Werte hat. Am höchsten sind daher die Speedups für das Infiniband-Netzwerk mit einer Geschwindigkeit von 10 Gbit/s. Am geringsten sind dagegen die Speedups für das nur 100 Mbit/s schnelle Fast-Ethernet-Netzwerk. Implementierung (Dbc) kommt am besten mit dem langsamen Fast-Ethernet-Netzwerk zurecht, da sie am besten in der Lage ist, die Übertragungszeiten durch Berechnungen zu überdecken. Ihr Speedup fällt daher nur von 51,0 auf 38,4. Für (PipeD) ist der Anteil der durch Berechnungen überdeckten Übertragungszeit dagegen geringer. Ihr maximaler Speedup fällt deshalb von 54,2 auf 26,9. Für (PipeD2) und (PipeD3) hingegen fällt der Speedup aufgrund der relativ großen Menge zu übertragender Daten von ca. 41 auf unter 9. Die Speedup-Kurven für das Gigabit-Ethernet-Netzwerk mit einer Geschwindigkeit von 1 Gbit/s ordnen sich erwartungsgemäß zwischen den Kurven für das Infiniband- und das Fast-Ethernet-Netzwerk ein. Interessant am Verlauf dieser Kurven ist vor allem die sprunghafte Verschlechterung des Speedups der Implementierungen (PipeD2) und (PipeD3) bei Erhöhung der Prozessoranzahl von 40 auf 48, der vermutlich dadurch zu erklären ist, daß ab dieser Prozessorzahl die Sweep-Phase nicht mehr ausreichend lang ist, um die Übertragungszeit für die nach der Initialisierungsphase versandten Daten zu überdecken. Für das Testproblem MEDAKZO unterscheiden sich die Speedup-Werte für die verschiedenen Netzwerke im Gegensatz zu BRUSS2D-MIX kaum. Da es auch zwischen den verschiedenen Implementierungsvarianten kaum Unterschiede gibt, liegt die Vermutung nahe, daß aufgrund der geringen Systemgröße die für die Stufenberechnung notwendige Kommunikation nur eine untergeordnete Rolle spielt und daß stattdessen die globale Kommunikationsoperation `MPI_Allreduce()`, die in allen Implementierungen innerhalb der Schrittweitenkontrolle mit einer sehr kleinen Nachrichtengröße aufgerufen wird, die Skalierbarkeit begrenzt.

Jülicher Multiprozessor (JUMP)

Der Supercomputer JUMP besteht aus mehreren 32-Wege-SMP-Knoten des Typs IBM p690. Aufgrund ihrer schlechten Skalierbarkeit konnten die allgemeinen Implementierungen nur auf bis zu 32 Prozessoren

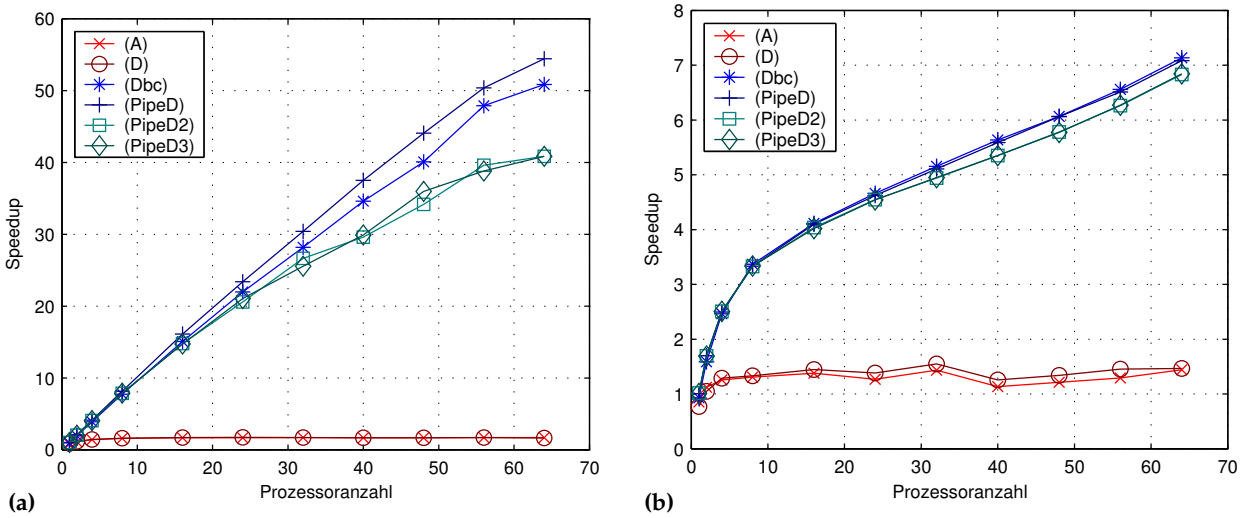


Abb. 4.20: Speedups der parallelen spezialisierten Implementierungen für verteilten Adreßraum im Vergleich zu den Implementierungen (A) und (D) auf einem Opteron-SMP-Cluster bei Verwendung eines Infiniband-Netzwerkes und MPICH 1.2.5. **(a)** BRUSS2D-MIX, DOPRI5(4), $N = 1000$, $H = 4,0$, **(b)** MEDAKZO, DOPRI8(7), $N = 30\,000$, $H = 10^{-4}$.

innerhalb eines Knotens ausgeführt werden. Alle für die allgemeinen Implementierungen für BRUSS2D-MIX und DOPRI5(4) gemessenen Speedup-Werte lagen deutlich unterhalb von 4 (vgl. Abschnitt 3.3.4). Die spezialisierten Implementierungen können hingegen sowohl für BRUSS2D-MIX als auch für MEDAKZO sehr gute Speedup-Werte erreichen (siehe Abb. 4.23). Für BRUSS2D-MIX und DOPRI5(4) erreicht bei Verwendung von 64 Prozessoren Implementierung (PipeD) den höchsten Speedup von 55,4. Die Implementierungen (PipeD2) und (PipeD3) sind in diesem Fall im Vergleich zu den übrigen spezialisierten Implementierungen am langsamsten. Sie erreichen nur Speedups von 39,7 bzw. 40,5. Implementierung (Dbc) erreicht mit 64 Prozessoren einen Speedup von 46,1. Da sie nicht der Einschränkung unterliegt, daß jedem Prozessor mindestens $2s$ Blöcke zugeordnet werden müssen, wurden für diese Implementierung Messungen für bis zu 128 Prozessoren durchgeführt, wobei ein Speedup von 86,0 erzielt wurde. Für MEDAKZO und DOPRI8(7) können aufgrund der größeren Anzahl von Blöcken 128 Prozessoren für alle Implementierungen genutzt werden. Bei Ausführung auf bis zu 32 Prozessoren eines Knotens unterscheiden sich die Speedups aller Implementierungen nur geringfügig. Sie betragen für 32 Prozessoren zwischen 21,7 und 22,8. Bei weiterer Erhöhung der Prozessorzahl erreichen jedoch die Implementierungen (PipeD2) und (PipeD3) deutlich höhere Speedups als (PipeD) und (Dbc). Für 128 Prozessoren betragen die für (PipeD2) und (PipeD3) erzielten Speedup-Werte 73,9 bzw. 72,6. (PipeD) und (Dbc) erreichen für diese Prozessoranzahl Speedups von 62,0 bzw. 60,4.

Sun Fire 6800

Die Sun Fire 6800 ist ein SMP-System mit 24 UltraSPARC-III-Prozessoren, von denen bis zu 8 für die Untersuchung der parallelen Implementierungsvarianten eingesetzt werden konnten. Die allgemeinen Implementierungen konnten in den in Abschnitt 3.3.4 durchgeführten Experimenten für die betrachtete Prozessorzahl ähnlich gute Speedups wie auf der Cray T3E-1200 von bis zu 6,6 für BRUSS2D-MIX erreichen. Abbildung 4.24 zeigt die Speedups der spezialisierten Implementierungen im Vergleich zu den Implementierungen (A) und (D) für zwei Experimente mit BRUSS2D-MIX und DOPRI5(4) sowie MEDAKZO und DOPRI8(7). Auf dieser Maschine verkleinern sich die Speedups der Implementierungen (A) und (D) durch Einbeziehung der spezialisierten Implementierungen besonders stark, da die sequentielle Implementierung (PipeD) eine erheblich bessere Laufzeit erreicht als die sequentiellen Implementierungen (A) und (D). Unter Verwendung der Laufzeit der sequentiellen Implementierung (PipeD) als Referenz beträgt der maximale Speedup der allgemeinen Implementierungen im Experiment mit BRUSS2D-MIX nur noch 4,7 anstelle von 6,6. Die Pipelining-Implementierungen können demgegenüber sogar superlineare Speedups zwischen 8,4 und 8,7 erreichen, wobei (PipeD) geringfügig schneller arbeitet als (PipeD2) und (PipeD3). Implementie-

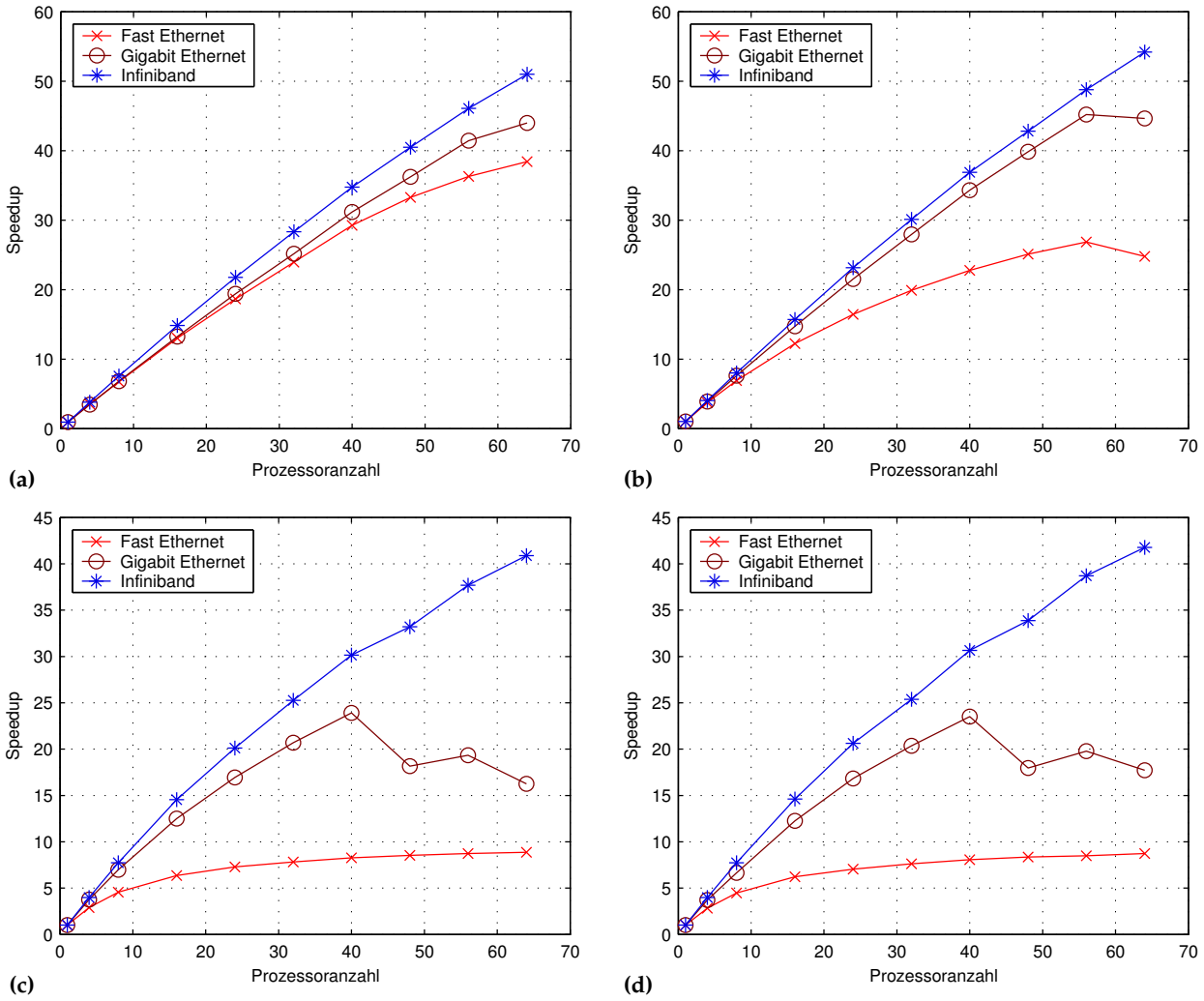


Abb. 4.21: Speedups der parallelen spezialisierten Implementierungen für verteilten Adreßraum auf einem Opteron-SMP-Cluster für BRUSS2D-MIX, DOPRI5(4), $N = 1000$ und $H = 4,0$ bei Verwendung von LAM/MPI 7.1.1 für verschiedene Verbindungsnetzwerke. **(a)** (Dbc), **(b)** (PipeD), **(c)** (PipeD2), **(d)** (PipeD3).

rung (Dbc) ist für kleine Prozessorzahlen aufgrund des schlechteren Lokalitätsverhaltens deutlich langsamer als die Pipelining-Implementierungen, kann ihren Speedup für steigende Prozessorzahl wegen der schrumpfenden Arbeitsmengen jedoch stärker als linear steigern, so daß sich ihr Speedup dem der Pipelining-Implementierungen annähert. Bei Verwendung von 8 Prozessoren erreicht sie einen Speedup von 8,0. Im Experiment mit MEDAKZO und DOPRI8(7) liegen die Speedups der allgemeinen Implementierungen bei 4,0 und 4,1. Auch für dieses Testproblem arbeiten die spezialisierten Implementierungen wesentlich schneller. Implementierung (Dbc) erzielt einen Speedup von 7,2. Am schnellsten arbeiten die Pipelining-Implementierungen, die Speedups zwischen 7,6 und 7,7 erreichen.

4.5 Implementierungen für gemeinsamem Adreßraum

Wie der vorige Abschnitt deutlich gemacht hat, kann eine beschränkte Zugriffsdistanz bei der Implementierung für verteilten Adreßraum zur Reduzierung des Kommunikationsaufwandes ausgenutzt werden. In den durchgeführten Laufzeitexperimenten auf verschiedenen Zielsystemen mit verteiltem Adreßraum konnten auf diese Weise gute bis sehr gute Speedups erreicht werden, wogegen die allgemeinen Imple-

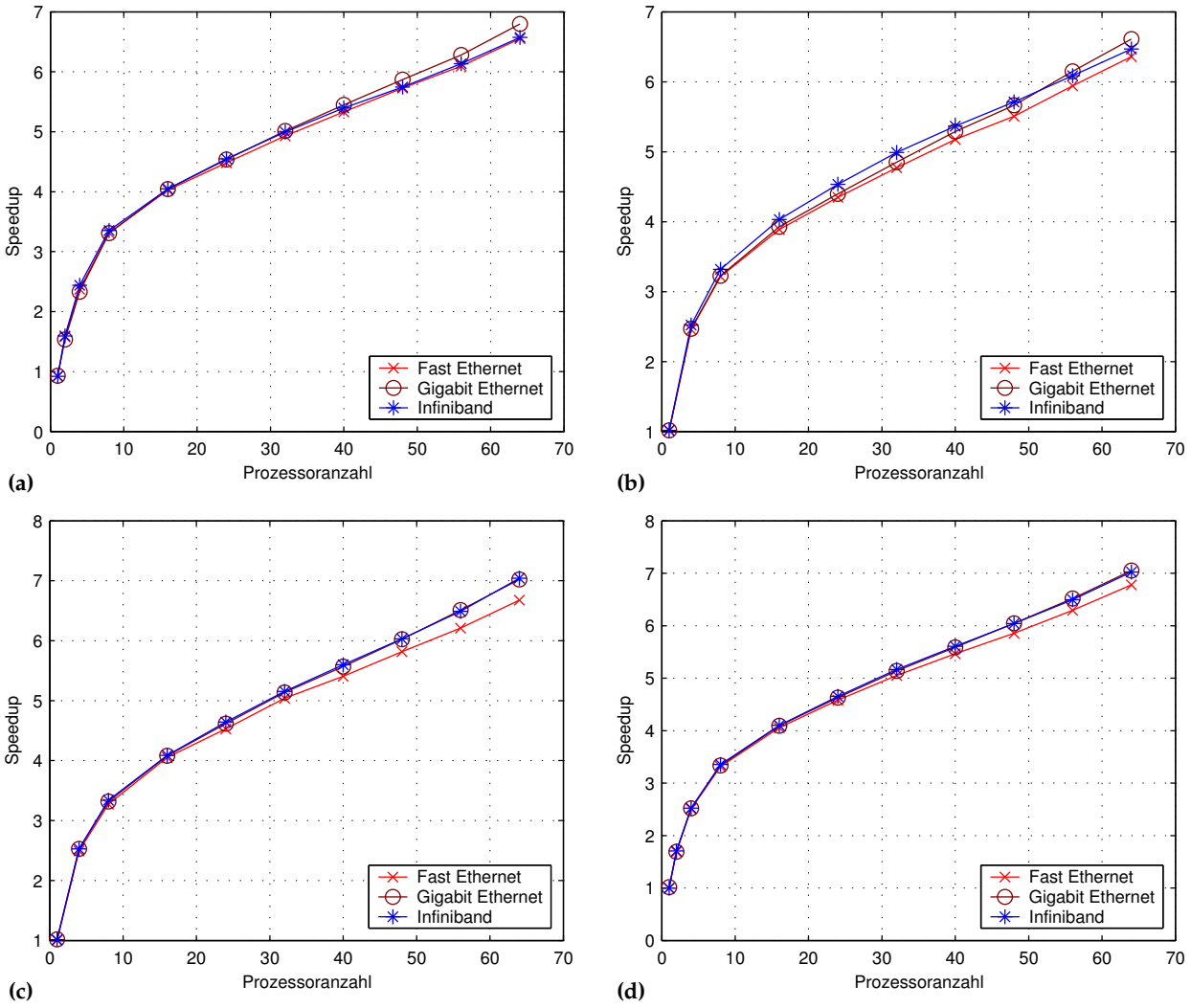


Abb. 4.22: Speedups der parallelen spezialisierten Implementierungen für verteilten Adreßraum auf einem Opteron-SMP-Cluster für MEDAKZO, DOPRI8(7), $N = 30\,000$ und $H = 10^{-4}$ bei Verwendung von LAM/MPI 7.1.1 für verschiedene Verbindungsnetzwerke. **(a)** (Dbc), **(b)** (PipeD), **(c)** (PipeD2), **(d)** (PipeD3).

mentierungen auf den meisten Systemen nur eine sehr schlechte Skalierbarkeit besitzen. In mehreren der durchgeführten Experimente konnte weiterhin beobachtet werden, daß für die spezialisierten Implementierungen auch das Lokalitätsverhalten Einfluß auf die Skalierbarkeit nimmt, während für die allgemeinen Implementierungen die Laufzeit überwiegend durch den Kommunikationsaufwand bestimmt wurde. So konnte beispielsweise auf einigen Systemen Implementierung (PipeD) aufgrund ihres besseren Lokalitätsverhaltens eine deutlich geringere Laufzeit als Implementierung (Dbc) erreichen, obwohl sie die gleichen Kommunikationsoperationen ausführt wie Implementierung (Dbc), jedoch weniger Berechnungen parallel zu den Datenübertragungen durchführen kann. Es ist daher zu erwarten, daß auch auf Parallelrechnern mit gemeinsamem Adreßraum, auf denen das Laufzeitverhalten entscheidend durch die Speicherzugriffslokalität beeinflußt wird, eine wesentliche Verbesserung der Skalierbarkeit möglich ist.

4.5.1 Realisierung der Synchronisation auf Rechnern mit gemeinsamem Adreßraum

Wie bereits in Abschnitt 3.4.1 diskutiert, können Programme für verteilten Adreßraum auf einen gemeinsamen Adreßraum übertragen werden, indem die Operationen zum Nachrichtenaustausch durch das Kopieren der zu übertragenden Daten innerhalb des gemeinsamen Adreßraumes nachgebildet werden. Die in

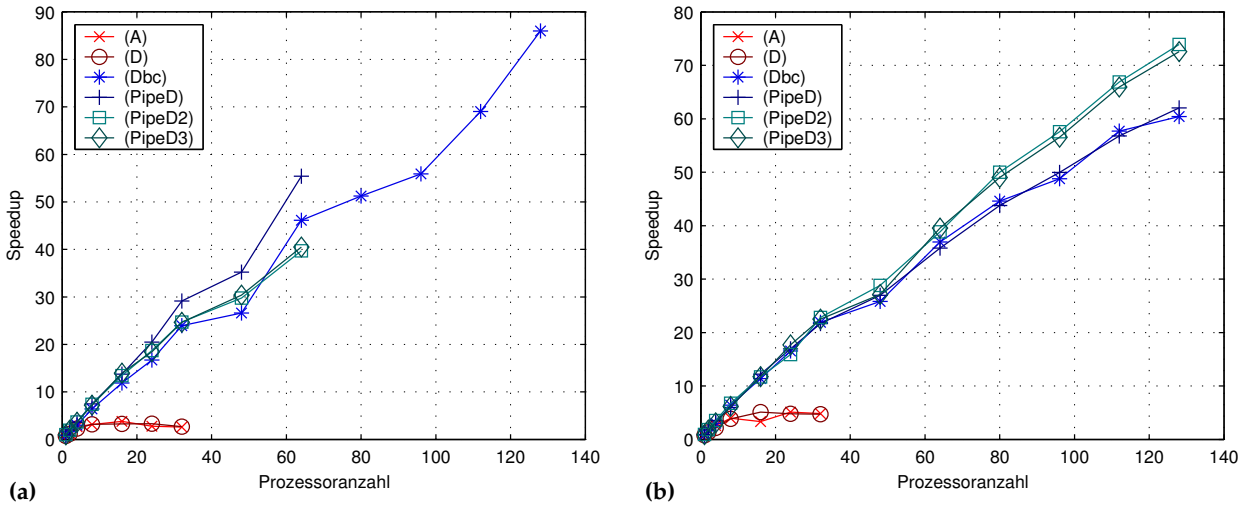


Abb. 4.23: Speedups der parallelen spezialisierten Implementierungen für verteilten Adreßraum im Vergleich zu den Implementierungen (A) und (D) auf dem Supercomputer JUMP. **(a)** BRUSS2D-MIX, DOPRI5(4), $N = 1000$, $H = 4,0$, **(b)** MEDAKZO, DOPRI8(7), $N = 30\,000$, $H = 10^{-4}$.

diesem Abschnitt vorgestellten Implementierungsvarianten nutzen jedoch ebenso wie die in Abschnitt 3.4.1 präsentierten Implementierungen gemeinsame Datenstrukturen, wodurch kein Kopieren von Daten erforderlich ist. Da der gemeinsame Adreßraum es allen beteiligten Threads ermöglicht, alle Vektordaten zu lesen und zu schreiben, muß stattdessen eine geeignete Synchronisation der Threads durchgeführt werden. Für die Realisierung der Implementierungen für gemeinsamen Adreßraum legen wir die gleiche blockweise Datenverteilung wie für die Implementierungen für verteilten Speicher zugrunde.

Unter diesen Gegebenheiten können wir die Implementierungen (Dbc) und (PipeD) auf gemeinsamen Adreßraum übertragen, indem wir den Pseudocode aus den Abb. 4.10 bzw. 4.13 unverändert übernehmen und lediglich die Makros zur Realisierung der blockweisen Kommunikation wie in Abb. 4.25 anpassen. Dazu werden die nichtblockierenden Einzeltransferoperationen durch eine auf Mutex-Variablen basierende Strategie ersetzt. Diese ordnet jedem Block, auf den ein Nachbarprozessor lesend zugreifen möchte, eine Mutex-Variable zu, die z. B. am Anfang jedes Zeitschrittes durch den Prozessor, der diesen Block später berechnet, gesperrt wird. Sofort nachdem die Berechnung des Blocks fertiggestellt wurde, gibt der jeweilige Prozessor diesen Block frei. Will ein Prozessor auf einen Block eines Nachbarprozessors zugreifen, muß er diesen zuvor sperren. Da die Berechnung der von Nachbarprozessoren benötigten Blöcke aufgrund der gewählten Berechnungsreihenfolge in der Regel abgeschlossen ist, bevor auf diese Blöcke durch die Nachbarprozessoren zur Berechnung anderer Blöcke zugegriffen werden muß, enstehen im Regelfall keine Wartezeiten beim Sperren dieser Blöcke, so daß ein Overhead nur durch die zum Sperren und Freigeben der Blöcke ausgeführten Instruktionen entsteht. Wenn das initiale Sperren der Mutex-Variablen wie in den hier betrachteten Implementierungen am Anfang jedes Zeitschrittes durchgeführt wird, muß z. B. durch die Ausführung einer Barrier-Operation sichergestellt werden, daß kein Prozessor einen Block, dessen Mutex-Variable noch nicht initialisiert wurde, für Berechnungen nutzt. Die Ausführung einer Barrier-Operation zu diesem Zweck in jedem Zeitschritt könnte jedoch vermieden werden, indem man ausnutzt, daß z. B. Block J_{last} des Argumentvektors $\mathbf{w}_l$ nicht mehr benötigt wird, wenn Block $J_{\text{last}} + 1$ des Argumentvektors $\mathbf{w}_{l+1}$ zur Verfügung steht, oder umgekehrt daß Block J_{last} des Argumentvektors $\mathbf{w}_{l+1}$ erst dann benötigt werden kann, wenn Block J_{last} des Argumentvektors $\mathbf{w}_l$ freigegeben wurde.

Die Implementierungen (PipeD2) und (PipeD3) für verteilten Adreßraum unterscheiden sich nur darin, wann der Datenaustausch für die vom Nachfolgerprozessor zur Aktualisierung von η während der Schrittweitenkontrolle benötigten Blöcke von $\Delta\eta$ erfolgt. Bei einer Implementierung für gemeinsamen Adreßraum, die gemeinsame Datenstrukturen verwendet, ist kein solcher expliziter Datenaustausch erforderlich, da die betreffenden Vektorkomponenten vom Nachfolgerprozessor direkt gelesen werden können. Durch die zur Bestimmung der Approximation des lokalen Fehlers ϵ notwendige Synchronisation ist gleichzeitig sichergestellt, daß die Berechnung aller Komponenten von $\Delta\eta$ abgeschlossen ist, bevor die Aktualisierung

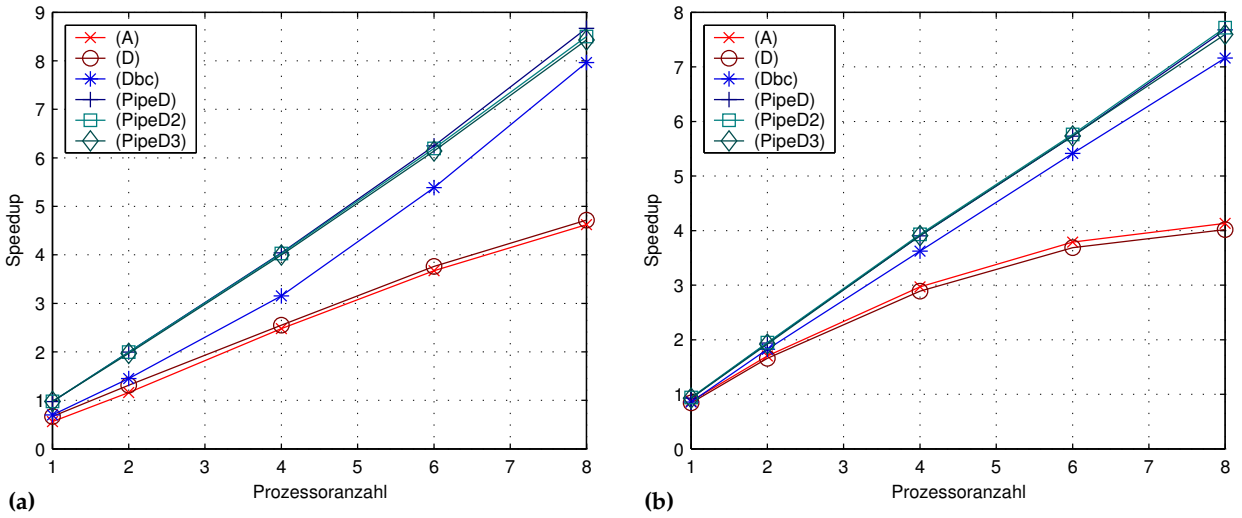


Abb. 4.24: Speedups der parallelen spezialisierten Implementierungen für verteilten Adreßraum im Vergleich zu den Implementierungen (A) und (D) auf einer Sun Fire 6800. **(a)** BRUSS2D-MIX, DOPRI5(4), $N = 750$, $H = 4,0$, **(b)** MED-AKZO, DOPRI8(7), $N = 30\,000$, $H = 10^{-4}$.

von η durchgeführt wird. Beide Implementierungsvarianten konvergieren daher bei der Übertragung auf einen gemeinsamen Adreßraum zu einer einzigen Implementierung, die wir als (PipeD2) bezeichnen wollen. Der Pseudocode dieser Implementierung ist in Abb. 4.26 dargestellt. Er berechnet die Blöcke der Vektoren in der gleichen Reihenfolge wie die beiden Implementierungen für verteilten Adreßraum. Es wird jedoch kein expliziter Datenaustausch durchgeführt. Stattdessen wird mittels einer einzigen Barrier-Operation, die vor der Finalisierungsphase ausgeführt wird, sichergestellt, daß die Berechnung aller für die Finalisierung benötigten Blöcke abgeschlossen wurde. Alternativ könnte man dies auch durch Verwendung von Mutex-Variablen realisieren.

Das Lokalitätsverhalten der spezialisierten Implementierungen für gemeinsamen Speicher entspricht im wesentlichen dem der jeweiligen Ausgangsimplementierung. Implementierung (Dbc) verarbeitet die Stufen sequentiell, wie dies auch in der Ausgangsimplementierung (D) der Fall ist. Im Unterschied zu (D) wird lediglich der Block J_{first} zu einem späteren Zeitpunkt berechnet und anstelle der Barrier-Operationen werden nun Mutex-Variablen zur Synchronisation benutzt. Charakteristisch für Implementierung (Dbc) ist daher ebenfalls die Wiederverwendung von Vektorblöcken der Größe $\lceil n/p \rceil + D - 1$. Für alle Pipelining-Implementierungen verbessert sich dagegen das Lokalitätsverhalten mit wachsender Prozessorzahl nur unwesentlich, da es durch die zeitliche Wiederverwendung des bereits relativ kleinen Arbeitsraumes eines Pipelining-Schrittes (4.2) geprägt ist, der nur durch die Stufenzahl und die Blockgröße bestimmt wird.

4.5.2 Laufzeitvergleich auf verschiedenen Zielsystemen

Die vorgestellten Implementierungsvarianten für gemeinsamen Adreßraum werden im folgenden bezüglich ihres Laufzeitverhaltens auf verschiedenen Zielsystemen mit unterschiedlicher Architektur untersucht. Dazu werden die Auswirkungen der Arbeitsmengen auf die Laufzeit auf einem Itanium-2-SMP-System sowie die Skalierbarkeit der Implementierungen auf einem Knoten des Supercomputers JUMP sowie auf einer Sun Fire 6800 betrachtet.

Auswirkung der Arbeitsmengen auf die Laufzeit

Wie bereits in den Abschnitten 2.4.4, 3.4.3 und 4.2.6 für die dort vorgestellten Implementierungen untersuchen wir auch für die spezialisierten Implementierungen für gemeinsamen Adreßraum als erstes die Auswirkung der Arbeitsmengen auf die Laufzeit, indem wir die Laufzeit bezüglich der Systemgröße und der Anzahl der Zeitschritte normieren und deren Verhalten bei wachsender Systemgröße analysieren. Als Zielsystem verwenden wir dazu wie bereits in den Experimenten zuvor ein Itanium-2-SMP-System, das

```
1:  #define INIT_STEP()                                   // Sperre alle noch nicht fertiggestellten Blöcke
2:  {
3:      for (l := 2; l ≤ s; l++)
4:      {
5:          lock(Block J_first von w_l);
6:          lock(Block J_last von w_l);
7:      }

8:      barrier();
9:  }

10: #define WAIT_FOR_PRED(m)        // Warte, bis der letzte Block von w_m des Vorgängers verfügbar ist
11: {
12:     lock(Block J_first − 1 von w_m);
13: }

14: #define WAIT_FOR_SUCC(m)        // Warte, bis der erste Block von w_m des Nachfolgers verfügbar ist
15: {
16:     lock(Block J_last + 1 von w_m);
17: }

18: #define FIRST_BLOCK_COMPLETE(m)                       // Der erste Block von w_m wurde berechnet
19: {
20:     if (m ≤ s) unlock(Block J_first von w_m);
21:     unlock(Block J_first − 1 von w_{m−1});
22: }

23: #define LAST_BLOCK_COMPLETE(m)                        // Der letzte Block von w_m wurde berechnet
24: {
25:     if (m ≤ s) unlock(Block J_last von w_m);
26:     unlock(Block J_last + 1 von w_{m−1});
27: }
```

Abb. 4.25: Makros zur Realisierung der blockweisen Kommunikation für gemeinsamen Adreßraum.

in Anhang A näher beschrieben wird. Wie bereits in Abschnitt 4.2.6 für die sequentiellen Varianten der spezialisierten Implementierungen betrachten wir auch hier einen Vergleich der Implementierung (PipeD) mit den allgemeinen Implementierungen (D) und (Dblock), wobei wir als Testprobleme BRUSS2D-MIX mit dem Verfahren DOPRI5(4) (Abb. 4.27) und MEDAKZO mit dem Verfahren DOPRI8(7) (Abb. 4.28) verwenden. Als Blockgröße benutzen wir für die spezialisierten Implementierungen ebenfalls $B = 2N$ für BRUSS2D und $B = 2$ für MEDAKZO. Für Implementierung (Dblock) wird wie in allen gleichartigen Experimenten zuvor eine Blockgröße von 5120 eingesetzt.

Bereits bei der Analyse der Arbeitsmengen der allgemeinen Implementierungen für gemeinsamen Adreßraum in Abschnitt 3.4.3 fiel auf, daß sich die Laufzeit der Implementierung (D) ab einer Vektorgröße von 128 KB sprungartig stark verschlechtert, was auf eine Ausrichtung der durch `malloc()` allokierten Speicherblöcke auf ein Vielfaches der Seitengröße von 16 KB zurückgeführt wurde. Durch das Zugriffsmuster der Implementierung (D), deren innerste Schleife schreibend auf Vektorelemente verschiedener Stufen mit gleichem Index zugreift, wird infolgedessen eine Erhöhung der Zahl der Konfliktfehlzugriffe hervorgerufen, was u. a. im Zusammenhang mit Interferenzen mit Speicherzugriffen auf den Programmcode zu dieser sprungartigen Laufzeiterhöhung führt. Die Laufzeit der Implementierung (PipeD) erhöht sich ebenfalls ab einer Vektorgröße von 128 KB um etwa den gleichen Betrag wie die Laufzeit der Implementierung (D). Die Ursache hierfür besteht darin, daß Implementierung (PipeD) zur Berechnung der Blöcke die gleiche Schleifenstruktur wie Implementierung (D) nutzt.

```
1: // Initialisierung der Pipeline

2: for (j = 1; j < s; j++)
3: {
4:     STAGE1(J_first + 2j − 1);
5:     for (i = 2; i ≤ j; i++) STAGE(J_first + 2j − i, i);

6:     STAGE1(J_first + 2j);
7:     for (i = 2; i ≤ j; i++) STAGE(J_first + 2j − i + 1, i);
8: }

9: // Diagonaler Sweep über die Argumentvektoren

10: for (j = J_first + 2s − 1; j ≤ J_last − 1; j++)
11: {
12:     STAGE1(j);
13:     for (i = 2; i ≤ s; i++) STAGE(j − i + 1, i);
14: }

15: // Warte, bis alle Prozessoren den Sweep abgeschlossen haben

16: barrier();

17: // Finalisierung der Pipeline

18: STAGE1(J_last);
19: for (i = 2; i ≤ s; i++) STAGE(J_last − i + 1, i);

20: STAGE1((J_last mod n_B) + 1);
21: for (i = 2; i ≤ s; i++) STAGE(J_last − i + 2, i);

22: for (i = 2; i ≤ s; i++)
23: {
24:     for (j = i; j ≤ s; j++) STAGE(((J_last + 2i − j − 2) mod n_B) + 1, j);
25:     for (j = i; j ≤ s; j++) STAGE(((J_last + 2i − j − 1) mod n_B) + 1, j);
26: }
```

Abb. 4.26: Parallele Implementierung (PipeD2) für gemeinsamen Adreßraum. Die Definition der Makros STAGE() und STAGE1() entspricht der in der sequentiellen Implementierung (PipeD) verwendeten Definition (siehe Abb. 4.2).

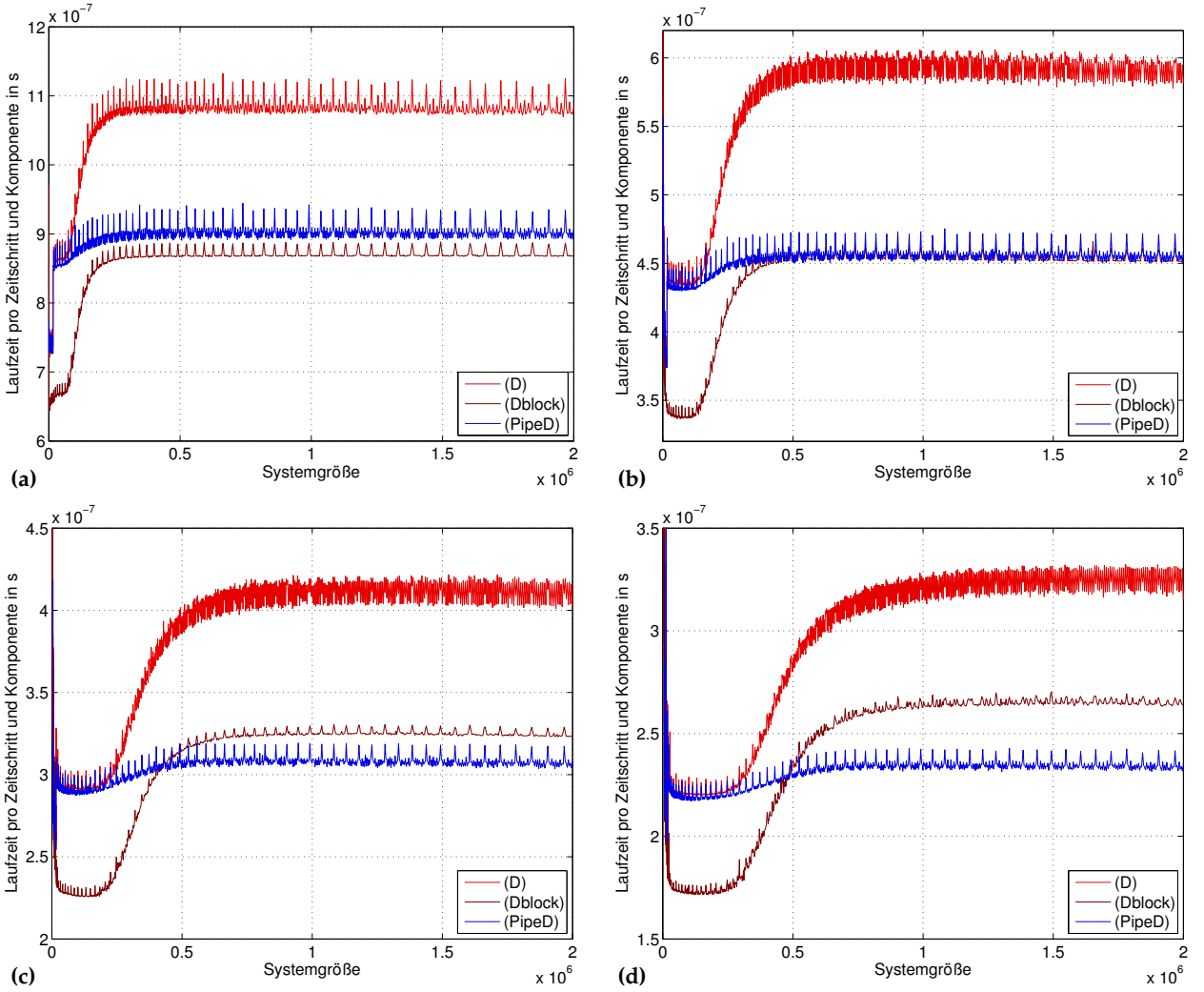


Abb. 4.27: Vergleich der Laufzeit der parallelen Implementierungen (D), (Dblock) und (PipeD) für gemeinsamen Adreßraum pro Zeitschritt und Komponente in Abhängigkeit von der Systemgröße auf einem Itanium-2-System für das Testproblem BRUSS2D-MIX und das Verfahren DOPRI5(4). **(a)** 1 Thread, **(b)** 2 Threads, **(c)** 3 Threads und **(d)** 4 Threads.

Auch bei der parallelen Ausführung sind die Laufzeiten der Implementierungen (D) und (PipeD) für kleine Problemgrößen sehr ähnlich. Sobald jedoch nicht mehr alle $(s+3)$ Vektorblöcke der Größe $\lceil n/p \rceil + D - 1$ im L3-Cache gespeichert werden können, erhöht sich die Laufzeit der Implementierung (D) aufgrund der gestiegenen Anzahl von Cache-Fehlzugriffen sehr stark, während die Laufzeit der Implementierung (PipeD) nur geringfügig ansteigt. Für kleine Systemgrößen sind daher auch die Speedups dieser beiden Implementierungen nahezu identisch. Ist die Systemgröße jedoch so hoch, daß die $(s+3)$ Vektorblöcke der Größe $\lceil n/p \rceil + D - 1$ aus dem L3-Cache herausfallen, kann sich die Laufzeit der Implementierung (D) nicht im gleichen Maße verbessern wie die Laufzeit der Implementierung (PipeD), so daß (PipeD) in diesem Fall wesentlich höhere Speedups erreicht als Implementierung (D).

Implementierung (Dblock) ist für kleine Systemgrößen immer deutlich schneller als die Implementierungen (D) und (PipeD). Passen nicht mehr alle $(s+3)$ Vektorblöcke der Größe $\lceil n/p \rceil + D - 1$ in den L3-Cache, steigt jedoch die Laufzeit von (Dblock) in ähnlichem Maße wie die Laufzeit der Implementierung (D) an. Für entsprechend hohe Systemgrößen ist daher (PipeD) auch bei paralleler Ausführung in den meisten Fällen deutlich schneller als (Dblock). Nur bei der Ausführung der parallelen Implementierungen mit nur einem Thread mit dem Testproblem BRUSS2D-MIX ist (Dblock) geringfügig schneller. Dies ist jedoch auf den sprunghaften Anstieg der Laufzeit von (PipeD) ab einer Vektorgröße von 128 KB zu-

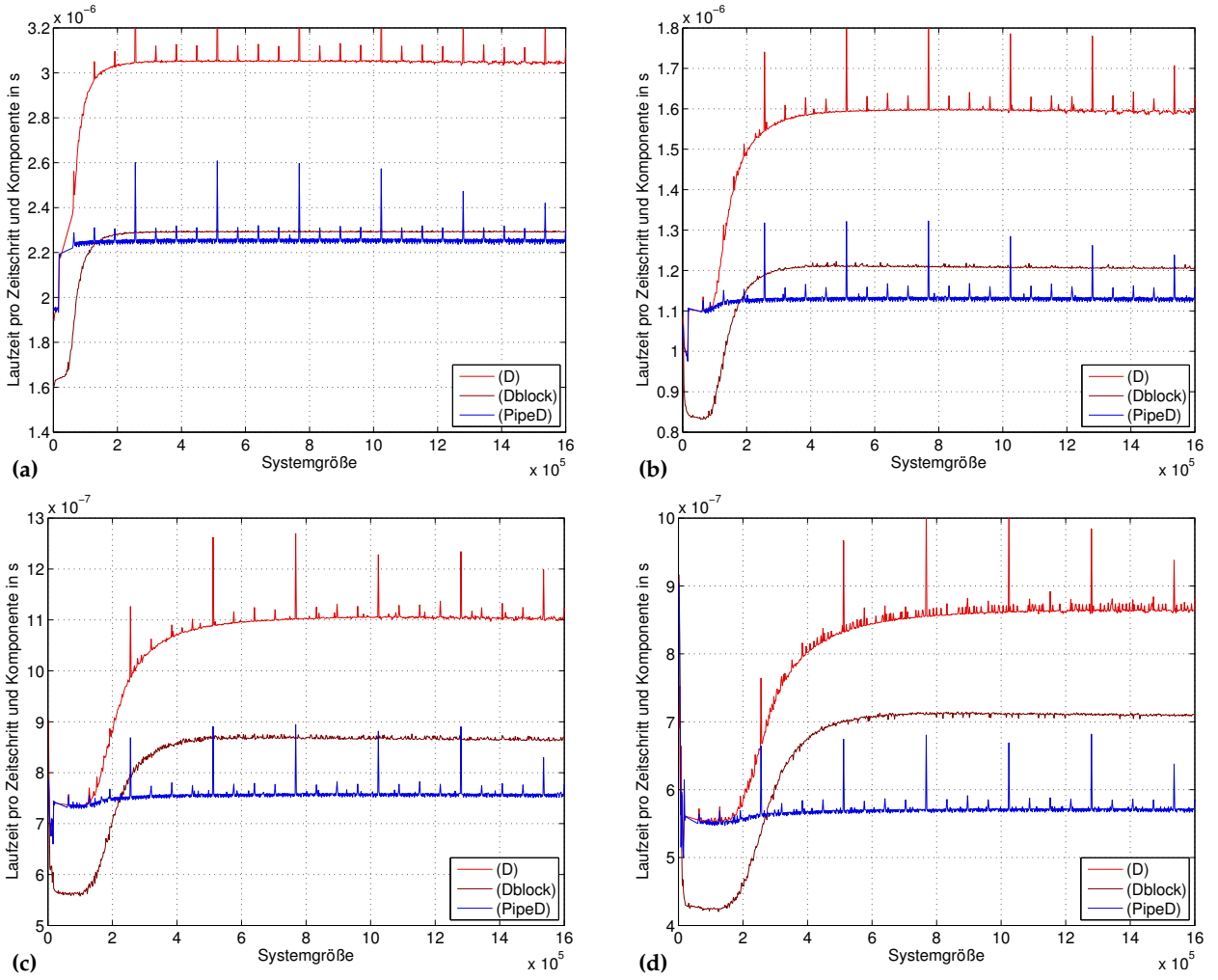


Abb. 4.28: Vergleich der Laufzeit der parallelen Implementierungen (D), (Dblock) und (PipeD) für gemeinsamen Adreßraum pro Zeitschritt und Komponente in Abhängigkeit von der Systemgröße auf einem Itanium-2-System für das Testproblem MEDAKZO und das Verfahren DOPRI8(7). **(a)** 1 Thread, **(b)** 2 Threads, **(c)** 3 Threads und **(d)** 4 Threads.

rückzuführen. Denn verwendet man dagegen die sequentiellen Implementierungsvarianten, tritt bei dieser Vektorgröße ein Laufzeitanstieg nur in geringem Maße auf, und Implementierung (PipeD) kann dadurch signifikant geringere Laufzeiten als (Dblock) erreichen. Durch die Ausführung auf mehreren Prozessoren verkleinern sich jedoch die durch den Lauf über die Stufen hervorgerufenen Arbeitsmengen der Implementierung (Dblock), da die Komponenten der Argumentvektoren auf die teilnehmenden Prozessoren aufgeteilt werden. Dementsprechend steigt die Systemgröße, ab der Implementierung (PipeD) schneller arbeitet als (Dblock), mit zunehmender Prozessorzahl an.

Skalierbarkeit

Zur Untersuchung der Skalierbarkeit der spezialisierten Implementierungen für gemeinsamen Adreßraum wurden Experimente auf einem IBM-p690-Knoten des Supercomputers JUMP sowie einer Sun Fire 6800 durchgeführt. Als Testprobleme wurden dazu wiederum BRUSS2D-MIX mit dem Verfahren DOPRI5(4) und MEDAKZO mit dem Verfahren DOPRI8(7) verwendet. Die Experimente wurden unter den gleichen Bedingungen wie die Experimente in Abschnitt 4.4.4 und 3.4.3 durchgeführt. Auch hier verwenden wir als Referenz für die Speedup-Berechnung die schnellste sequentielle Variante der jeweils betrachteten Implementierungen.

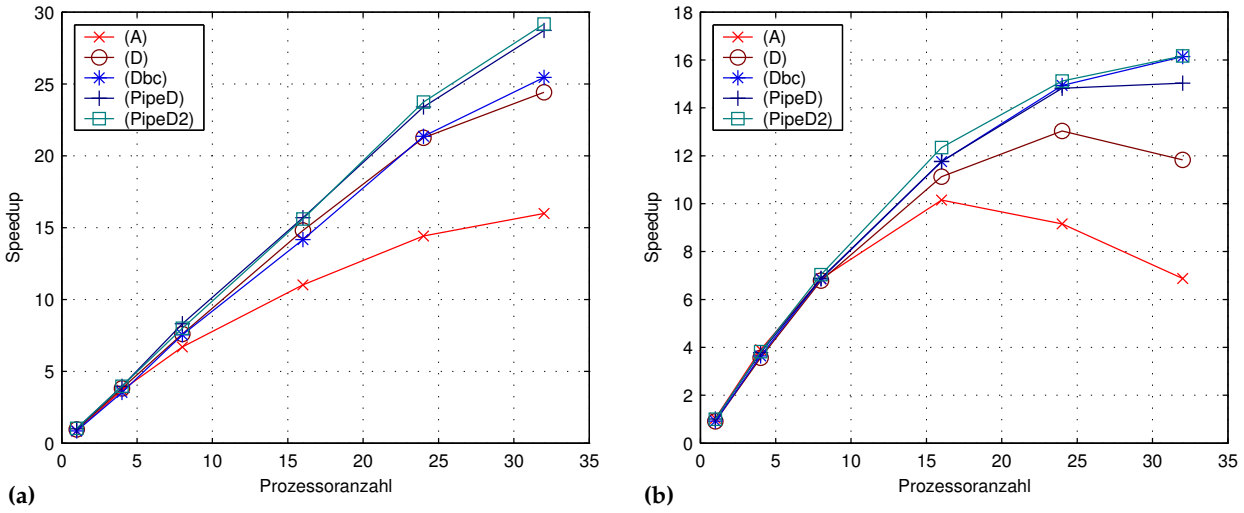


Abb. 4.29: Speedups der parallelen spezialisierten Implementierungen für gemeinsamen Adreßraum im Vergleich zu den Implementierungen (A) und (D) auf dem Supercomputer JUMP. **(a)** BRUSS2D-MIX, DOPRI5(4), $N = 1000$, $H = 4,0$, **(b)** MEDAKZO, DOPRI8(7), $N = 30\,000$, $H = 10^{-4}$.

IBM p690. Die auf einem Knoten des Supercomputers JUMP des Typs IBM p690 mit 32 Prozessorkernen für die Implementierungen (A), (D), (Dbc), (PipeD) und (PipeD2) erzielten Speedup-Werte zeigt Abb. 4.29. Für BRUSS2D-MIX und DOPRI5(4) (Abb. 4.29 (a)) wird die Skalierbarkeit maßgeblich durch das Lokalitätsverhalten bestimmt. Dies wird dadurch deutlich, daß die Implementierungen (D) und (Dbc) sehr ähnliche Speedups von 24,4 und 25,4 erreichen, obwohl sie unterschiedliche Synchronisationsmechanismen nutzen. Aufgrund des unterschiedlichen Lokalitätsverhaltens sind beide Implementierungen wesentlich schneller als Implementierung (A), die nur einen Speedup von 16,0 erreicht, sie sind jedoch auch deutlich langsamer als die Implementierungen (PipeD) und (PipeD2), die einen Speedup von 28,7 bzw. 29,2 erreichen. Die Speedups von (PipeD) und (PipeD2) liegen ebenfalls sehr dicht beieinander, obwohl unterschiedliche Synchronisationsmechanismen genutzt werden, weil beide Implementierungen ein sehr ähnliches Lokalitätsverhalten besitzen. Für MEDAKZO und DOPRI8(7) (Abb. 4.29 (b)) ist dagegen ein stärkerer Einfluß des Synchronisationsaufwandes erkennbar. Zwar ist auch hier aufgrund des unterschiedlichen Lokalitätsverhaltens Implementierung (D) schneller als Implementierung (A), jedoch entspricht das Skalierbarkeitsverhalten der Implementierung (Dbc) eher dem der übrigen spezialisierten Implementierungen, (PipeD) und (PipeD2), mit ähnlichem Synchronisationsaufwand als dem der korrespondierenden allgemeinen Implementierung (D) mit ähnlichem Lokalitätsverhalten. Die für MEDAKZO erzielten Speedups sind geringer als für BRUSS2D-MIX. Die allgemeinen Implementierungen (A) und (D) erreichen lediglich Speedups von 6,9 bzw. 11,8. Die höchsten Speedups von 16,1 bzw. 16,2 erreichen die Implementierungen (Dbc) und (PipeD2). Für Implementierung (PipeD) wurde ein Speedup von 15,0 gemessen.

Vergleicht man die von den Implementierungen für gemeinsamen Adreßraum erzielten Speedups mit den in den Abschnitten 4.4.4 und 3.3.4 für die Implementierungen für verteilten Adreßraum berichteten Speedup-Werten, so konnte durch die Nutzung des gemeinsamen Adreßraumes zwar für die allgemeinen Implementierungen (A) und (D) wie erwartet eine Verbesserung der Skalierbarkeit erreicht werden, für die spezialisierten Implementierungen wurden dagegen bei jeweils gleicher Prozessorzahl von den MPI-Implementierungen teilweise höhere Speedups erzielt als von den POSIX-Threads-Implementierungen. Dies trifft insbesondere auf die Experimente mit dem Testproblem MEDAKZO zu. Es ist daher zu vermuten, daß es aufgrund der NUMA-Architektur dieses Systems vorteilhaft ist, wenn jeder Kontrollfluß hauptsächlich auf von ihm selbst allokierte Speicherbereiche zugreift, da sich diese im günstigsten Fall in einem dem ausführenden Prozessor zugeordneten Speichermodul befinden und deshalb geringe Speicherzugriffszeiten ermöglichen. Gegenüber gemeinsam genutzten Datenstrukturen, wie sie in den vorgestellten Implementierungen genutzt werden, erfordern solche verteilten Datenstrukturen zwar einen expliziten Datenaustausch, der mit einem Overhead verbunden ist, bei Verwendung gemeinsam genutzter Datenstrukturen kann jedoch häufig nur ein einziger Prozessor auf ein lokales Speichermodul zugreifen. Alle übrigen Prozessoren

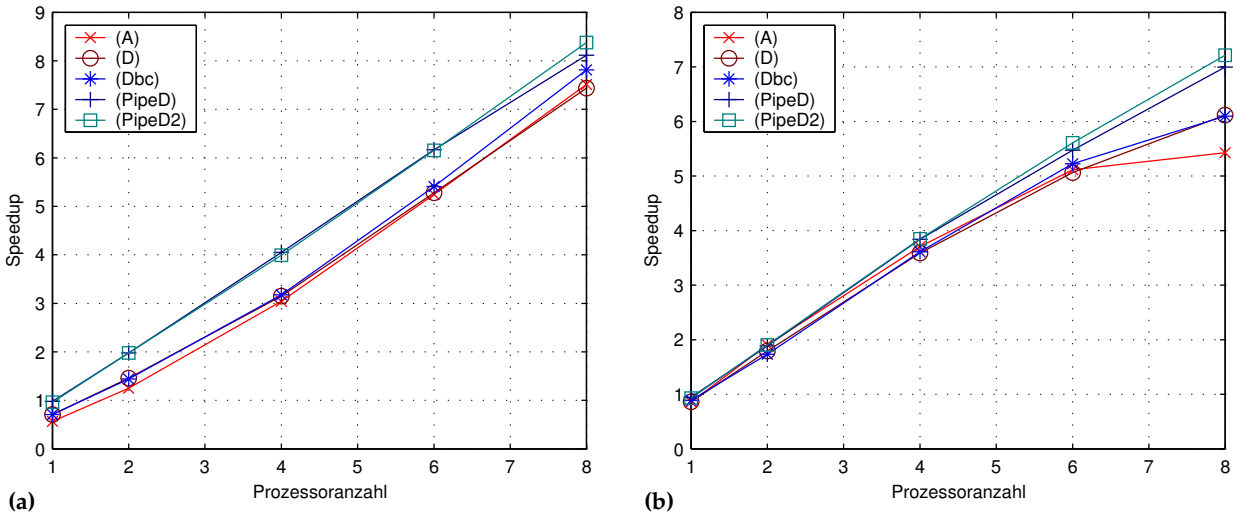


Abb. 4.30: Speedups der parallelen spezialisierten Implementierungen für gemeinsamen Adreßraum im Vergleich zu den Implementierungen (A) und (D) auf einer Sun Fire 6800. **(a)** BRUSS2D-MIX, DOPRI5(4), $N = 750$, $H = 4,0$, **(b)** MEDAKZO, DOPRI8(7), $N = 30\,000$, $H = 10^{-4}$.

müssen eine Vielzahl von Remote-Speicherzugriffen ausführen, was zu einer höheren Belastung des Speichersystems und zu höheren Speicherzugriffszeiten führt.

Sun Fire 6800. Die Ergebnisse der auf einer Sun Fire 6800 durchgeführten Experimente zeigt Abb. 4.30. Auf dieser Maschine wird die Laufzeit für beide Testprobleme wesentlich durch das Lokalitätsverhalten bestimmt. Deshalb bieten sowohl für BRUSS2D-MIX als auch für MEDAKZO die Pipelining-Implementierungen die beste Skalierbarkeit, wobei ihre Speedup-Kurve bis zur betrachteten Anzahl von 8 Prozessoren sehr geradlinig verläuft. Für BRUSS2D-MIX und DOPRI5(4) erreichen (PipeD) und (PipeD2) sogar superlineare Speedups von 8,1 bzw. 8,4. Demgegenüber beträgt ihr Speedup für MEDAKZO nur 7,0 bzw. 7,2. Der Verlauf der Speedup-Kurven der übrigen Implementierungen ist für die beiden Testprobleme sehr unterschiedlich. Für BRUSS2D-MIX liefern die Implementierungen (A), (D) und (Dbc) auf einem Prozessor Speedup-Werte zwischen 0,6 und 0,7. Ihre Speedups steigern sich jedoch stärker als linear, so daß diese Implementierungen auf 8 Prozessoren Speedups von 7,5, 7,4 und 7,8 erreichen und sich somit der Effizienz der Pipelining-Implementierungen stetig annähern. Jedoch ist erkennbar, daß Implementierung (Dbc) ihren Speedup für mehr als 6 Prozessoren meßbar stärker steigern kann als die allgemeinen Implementierungen (A) und (D). Das Lokalitätsverhalten bestimmt also die Skalierbarkeit nicht allein, sondern es gibt auch einen meßbaren Einfluß des Synchronisationsaufwandes. Für MEDAKZO und DOPRI8(7) erreichen die Implementierungen (A), (D) und (Dbc) zwar ebenfalls gute Speedup-Werte, die Effizienz der Implementierung sinkt jedoch mit wachsender Prozessorzahl deutlich ab. Die Speedup-Werte aller drei Implementierungen liegen für bis zu 6 Prozessoren dicht zusammen. Für 8 Prozessoren verschlechtert sich die Effizienz der Implementierung (A) stärker als die Effizienz der Implementierungen (D) und (Dbc), so daß in diesem Fall (A) nur einen Speedup von 5,4 erreicht, während (D) und (Dbc) beide einen Speedup von 6,1 erzielen.

Im Vergleich zu den Implementierungen für verteilten Adreßraum wurden auch auf dieser Maschine für die spezialisierten Implementierungen teilweise höhere Speedups durch die MPI-Implementierungen erzielt als durch die POSIX-Threads-Implementierungen. Die Laufzeitunterschiede sind jedoch erheblich geringer als auf dem IBM-p690-System. Eine mögliche Erklärung hierfür sind die weniger großen Unterschiede in den Speicherzugriffszeiten zwischen lokalem und entferntem Speicher auf der Sun Fire 6800.

4.6 Hybride Implementierungen für SMP-Cluster

Zwar haben wir im vorangegangenen Abschnitt die Beobachtung gemacht, daß in einigen Fällen die MPI-Versionen der spezialisierten Implementierungen auf Parallelrechnern mit gemeinsamem Adreßraum hö-

here Speedups erreichen konnten als die entsprechenden mit POSIX Threads implementierten Versionen, da es sich jedoch bei POSIX Threads um das native Programmiermodell für diese Art von Parallelrechner handelt, müssen wir daraus die Schlußfolgerung ziehen, daß die POSIX-Threads-Versionen nicht optimal für die betrachteten Rechnersysteme angepaßt waren, nicht jedoch, daß MPI generell für diese Implementierungen Vorteile gegenüber POSIX Threads bieten würde. Für das Erreichen einer hohen Skalierbarkeit auf SMP-Clustersystemen, die einen Verbund mehrerer Parallelrechner mit gemeinsamem Adreßraum darstellen, die über ein Kommunikationsnetzwerk Nachrichten austauschen können, erscheint es deshalb auch für die auf eine beschränkte Zugriffsdistanz spezialisierten Implementierungen erfolgversprechend, ein hybrides Programmiermodell einzusetzen, das MPI und POSIX Threads miteinander kombiniert. Dieser Ansatz wurde bereits für die allgemeinen Implementierungen erfolgreich genutzt (vgl. Abschnitt 3.5).

In der von Behrend (2005) der Universität Bayreuth vorgelegten Bachelor-Thesis wurden neben Varianten der allgemeinen Implementierungen (A) und (D) auch spezialisierte Implementierungsvarianten betrachtet, die eine beschränkte Zugriffsdistanz ausnutzen. Wie im Fall der allgemeinen Implementierungsvarianten gestatten auch die spezialisierten Implementierungsvarianten die Erzeugung einer Anzahl von MPI-Prozessen, von denen jeder aus einer vorgegebenen Anzahl von Threads besteht. Die Verteilung der Iterationen der Schleifen über die Systemdimension auf alle beteiligten Threads erfolgt wie im allgemeinen Fall blockweise, wobei Threads, die zu demselben MPI-Prozeß gehören, die benötigten Vektoren als gemeinsame Datenstrukturen nutzen. Zur Durchführung des Nachrichtenaustausches mittels MPI wurden verschiedene Vorgehensweisen betrachtet:

1. Da die spezialisierten Implementierungen einen Nachrichtenaustausch nur zwischen benachbarten Kontrollflüssen erfordern, benötigen nur je zwei Threads eines MPI-Prozesses Daten eines Threads, der zu einem anderen MPI-Prozeß gehört. Eine Möglichkeit zur Realisierung des Nachrichtenaustausch besteht deshalb darin, daß diese beiden Threads eines MPI-Prozesses selbst den Datenaustausch mit dem jeweils benachbarten MPI-Prozeß durch direkten Aufruf der entsprechenden MPI-Operationen durchführen. Da der MPI-Standard nicht garantiert, daß mehrere Threads gleichzeitig MPI-Operationen ausführen dürfen, muß dabei durch wechselseitigen Ausschluß eine Sequentialisierung der MPI-Operationen erzwungen werden.

2. Ein Nachteil dieses Vorgehens ist, daß diejenigen Threads, die MPI-Operationen ausführen, bei einer gleichförmigen blockweisen Verteilung eine höhere Laufzeit erfordern als die übrigen Threads. Eine alternative Vorgehensweise, die zumindest den Instruktionsoverhead der MPI-Operationen innerhalb der Arbeitsthreads vermeidet, besteht darin, einen speziellen Kommunikationsthread einzusetzen. MPI-Operationen werden dann ausschließlich von diesem Kommunikationsthread durchgeführt. Der dadurch entstehende Instruktionsoverhead kann somit im Idealfall durch den Scheduler des Betriebssystems auf die vorhandenen Prozessoren verteilt werden. Die Wartezeiten, die durch das Warten auf das Eintreffen von Nachrichten entstehen, können auf diese Weise jedoch nicht vermieden werden, so daß nach wie vor die beiden Threads am Rand des Datenbereiches eines MPI-Prozesses oft eine höhere Laufzeit benötigen, als die Rechenzeit der übrigen Threads beträgt.

Diesen Vorgehensweisen folgend wurden Varianten der Implementierungen (Dbc), (PipeD), (PipeD2) und (PipeD3) realisiert. Eine Untersuchung auf dem Supercomputer JUMP sowie auf einem Opteron-SMP-Cluster zeigte jedoch keine wesentliche Laufzeitverbesserung durch die Verwendung der betrachteten Implementierungsvarianten, da nach wie vor die Laufzeit der MPI-Operationen einen entscheidenden Einfluß auf die Programmlaufzeit ausübt. Um zu vermeiden, daß diejenigen Threads, die auf Daten eines anderen MPI-Prozesses angewiesen sind, nicht die übrigen Arbeitsthreads ihres eigenen MPI-Prozesses ausbremsen, könnte man versuchen, Verfahren zur dynamischen Lastbalancierung einzusetzen. Diese könnten eine gleichmäßige Ausnutzung aller verfügbaren Prozessorressourcen und eine dynamische Anpassung an möglicherweise variierende Nachrichtenübertragungszeiten ermöglichen.

4.7 Dynamische Lastbalancierung

In Abschnitt 3.6 wurde untersucht, in welchen Fällen der Einsatz dynamischer Lastbalancierungsverfahren erfolgreich zur Verbesserung der Laufzeit allgemeiner paralleler Implementierungen eingebetteter Runge-Kutta-Verfahren eingesetzt werden kann. Dazu wurden zunächst ausschließlich Parallelrechner mit ge-

meinsamem Adreßraum betrachtet, da sich hier eine Lastbalancierung mit geringem Overhead erreichen läßt. Die in diesem Zusammenhang durchgeführten Experimente mit verschiedenen Lastbalancierungsstrategien haben gezeigt, daß der Einsatz von Techniken zur dynamischen Lastbalancierung in vielen Fällen zu einer Laufzeitverbesserung führen kann. Nutzt man spezielle Maschineninstruktionen zur Minimierung des Overheads aus, können z. T. sogar für Probleme, deren Funktionsauswertungskosten nahezu gleichmäßig über die einzelnen Komponenten verteilt sind, Laufzeitverbesserungen erzielt werden. Unterscheiden sich die Funktionsauswertungskosten der einzelnen Komponenten sehr stark, ist eine Lastbalancierung zur effizienten Lösung des betreffenden Problems sogar zwingend erforderlich.

Es liegt daher nahe, daß dynamische Lastbalancierungsverfahren auch für die Realisierung spezialisierter Implementierungen erfolgreich eingesetzt werden könnten. Speziell zur Realisierung hybrider Implementierungen für Cluster von SMP-Systemen erscheint der Einsatz dynamischer Lastbalancierungsverfahren vielversprechend (vgl. Abschnitt 4.6). Durch die Annahme einer beschränkten Zugriffsdistanz ist es dabei nicht notwendig, wie im Fall der allgemeinen Implementierungen die Stufen sequentiell zu berechnen. Bricht man die sequentielle Berechnungsreihenfolge der Stufen auf, müssen jedoch die durch die Funktionsauswertung gegebenen Datenabhängigkeiten berücksichtigt werden. Dazu ist es erforderlich, die daraus resultierenden Abhängigkeiten zwischen Tasks auf geeignete Weise zu modellieren, indem die Tasks entweder erst dann dynamisch erzeugt werden, wenn alle ihre jeweiligen Abhängigkeiten erfüllt sind, oder indem die Tasks zwar bereits zu Beginn eines Zeitschrittes erzeugt werden, sie jedoch erst nach Erfüllung aller Abhängigkeiten den Status „ausführbar“ erhalten.

Die Strategie, nach der die ausführbaren Tasks abgearbeitet werden, bestimmt die Berechnungsreihenfolge der Vektorkomponenten. Werden die Tasks beispielsweise in FIFO-Reihenfolge (englisch: *first in, first out*) verarbeitet, führt dies dazu, daß auf dem Taskgraphen, der die Abhängigkeiten zwischen den Tasks beschreibt, eine Breitensuche durchgeführt wird. Daraus würde sich eine Berechnungsreihenfolge ergeben, die im wesentlichen eine sequentielle Berechnung der Stufen ähnlich Implementierung (Dbc) vornimmt. Wählt man dagegen eine LIFO-Reihenfolge (englisch: *last in, first out*) für das Scheduling der Tasks, erfolgt eine Tiefensuche auf dem Taskgraphen, die zu einer Berechnungsreihenfolge ähnlich dem Pipelining-Berechnungsschema führt. Die Scheduling-Strategie hat deshalb einen entscheidenden Einfluß auf das Lokalitätsverhalten.

Da im Unterschied zu den allgemeinen Implementierungen die Barrier-Synchronisation zwischen den Stufen entfällt, ist davon auszugehen, daß der Einsatz dynamischer Lastbalancierungsverfahren für die spezialisierten Implementierungen ein größeres Potential zur Laufzeitverbesserung bietet, als dies für die allgemeinen Implementierungen der Fall war. Denn durch die Modellierung der Abhängigkeiten wird es möglich, daß die beteiligten Kontrollflüsse alle Komponenten aller Stufen asynchron berechnen können und daß Wartezeiten durch die Ausführung von Barrier-Operationen oder durch das Warten auf die Berechnung benötigter Daten durch Nachbarprozessoren entfallen. Sofern es also gelingt, das Scheduling der Tasks weitestgehend wartezeitfrei und mit geringem Overhead zu realisieren, verspricht dieser Ansatz eine signifikante Verbesserung der Skalierbarkeit. Eine Implementierung von Techniken zur dynamischen Lastbalancierung, die eine beschränkte Zugriffsdistanz ausnutzen können, sowie eine Untersuchung des resultierenden Laufzeitverhaltens wurde jedoch bisher noch nicht durchgeführt, so daß diese Aussage momentan nicht durch experimentelle Ergebnisse bestätigt oder widerlegt werden kann.

4.8 Ansatz zur automatischen Spezialisierung

Die vorangegangenen Abschnitte dieses Kapitels haben sich damit beschäftigt, eine Laufzeitverbesserung für sequentielle und parallele Implementierungen eingebetteter Runge-Kutta-Verfahren durch die Ausnutzung einer spezifischen Eigenschaft des zu lösenden Problems, der beschränkten Zugriffsdistanz, zu erzielen. Dieser Ansatz hat sich als sehr erfolgreich erwiesen. Er kann dazu verwendet werden, das Lokalitätsverhalten sequentieller und paralleler Implementierungen zu verbessern, den Speicherplatzbedarf zu reduzieren, zu verwerfende Schritte frühzeitig zu erkennen sowie den Kommunikationsaufwand paralleler Implementierungen zu verringern. Somit konnte durch die Spezialisierung auf eine bestimmte Problemklasse eine wesentliche Verbesserung der Laufzeit bzw. der Skalierbarkeit erreicht werden. Ein alternativer Ansatz zur Realisierung effizienter paralleler Implementierungen besteht in der Entkoppelung der Stufen durch eine Spezialisierung bezüglich der Verfahrenskoeffizienten, wie sie u. a. von Iserles u. Nørsett

(1990) und Jackson u. Nørsett (1995) diskutiert wird. Die Spezialisierung – sei es bezüglich des Problems oder bezüglich der Verfahrenskoeffizienten – ist also ein erfolgreiches Konzept zur Parallelisierung und zur Verbesserung der Effizienz. Die manuelle Umsetzung dieses Konzeptes erfordert jedoch einen hohen Aufwand. Daher stellt sich die Frage, inwiefern eine automatische Ausnutzung spezifischer Eigenschaften des Problems oder des Verfahrens möglich ist.

Eine automatische Spezialisierung bezüglich der Verfahrenskoeffizienten ohne Berücksichtigung spezifischer Eigenschaften des Problems ist z. B. mittels der von Iserles u. Nørsett (1990) vorgeschlagenen Digraphmethode möglich und wurde von Petcu (1999) und Petcu u. Dragan (2000) in der Software *EpODE* realisiert. Zu diesem Zweck werden die Abhängigkeiten zwischen den Stufen durch einen gerichteten Graphen modelliert, anhand dessen die Stufenberechnungen durch ein statisches Scheduling-Verfahren auf die verfügbaren Prozessoren verteilt werden.

Gelingt es, auch die Abhängigkeiten der Funktionsauswertungen auf geeignete Weise zu modellieren, könnten statische Scheduling-Verfahren auch für eine problemspezifische Spezialisierung eingesetzt werden. Als Ausgangspunkt könnte ein Datenflußgraph ähnlich Abb. 2.6 dienen, der zunächst alle im allgemeinen Fall erlaubten Abhängigkeiten enthält. Die Spezialisierung erfolgt dann, indem nicht vorhandene Abhängigkeiten aus dem Graphen entfernt werden. Dies geschieht durch das Löschen der betroffenen Kanten und gegebenenfalls auch das Löschen von Knoten.

- **Spezialisierung bezüglich der Verfahrenskoeffizienten:** Die Verfahrenskoeffizienten treten ausschließlich als Gewichte in Summen auf, d. h., es wird jeweils ein Summand einer Summe gebildet, indem mit einem Verfahrenskoeffizienten multipliziert wird. Eine Spezialisierung bezüglich der Verfahrenskoeffizienten ist in folgenden Fällen möglich:

  1. Ein Koeffizient hat den Wert 1. Dann würde die Multiplikation mit diesem Verfahrenskoeffizienten das Ergebnis der Multiplikation nicht beeinflussen. Die Abhängigkeit zwischen diesem Koeffizienten und der Multiplikationsoperation in Form der betreffenden Kante kann also gelöscht werden.
  2. Ein Koeffizient hat den Wert 0. In diesem Fall beträgt das Ergebnis der Multiplikationsoperation unabhängig von den übrigen Faktoren 0. Das bedeutet, daß das Ergebnis dieser Multiplikationsoperation die nachfolgende Summation nicht mehr beeinflußt. Der Knoten des Graphen, der die Multiplikationsoperation repräsentiert, kann also zusammen mit sämtlichen ein- und ausgehenden Kanten gelöscht werden.

  Ergibt sich bei diesem Vorgehen die Situation, daß ein Knoten, der eine Multiplikations- oder Additionsoperation repräsentiert, nur höchstens eine eingehende Kante oder keine ausgehende Kante mehr besitzt, ist darauf entsprechend zu reagieren.

- **Spezialisierung bezüglich des Problems:** Unter der Voraussetzung, daß eine Funktionsauswertung $f_j(t, \mathbf{w})$ unabhängig von den konkreten Werten von t und $\mathbf{w}$ ausschließlich bestimmte Komponenten des Argumentvektors $\mathbf{w}$ benutzt, kann eine Spezialisierung bezüglich des Problems durchgeführt werden, indem für alle nicht benutzten Komponenten des Argumentvektors $\mathbf{w}$ die Kanten, welche die entsprechenden Abhängigkeiten beschreiben, aus dem Graphen entfernt werden.

Nach Durchführung dieser Spezialisierungsschritte erhält man einen Graphen, der den spezifischen Datenfluß für ein gegebenes Verfahren und ein gegebenes Problem beschreibt. Auf diesen Graphen könnte man nun ein statisches Scheduling-Verfahren anwenden, welches nach vorgegebenen Optimierungskriterien einen Ablaufplan generiert.

- **Parallelisierung**: Um die Berechnungen auf mehrere Prozessoren zu verteilen, muß das Scheduling-Verfahren gleichzeitig eine Partitionierung des Graphen erzeugen, so daß jedem Prozessor eine Partition zur Ausführung zugeordnet werden kann. Die Partitionierung sollte dabei im Idealfall derart erfolgen, daß jedem Prozessor der gleiche Berechnungsaufwand zugeordnet wird, so daß alle Prozessoren zum gleichen Zeitpunkt mit der Durchführung von Berechnungen beginnen können und ohne Unterbrechungen Berechnungen ausführen können, bis alle Prozessoren zum gleichen Zeitpunkt terminieren.

  Zur Reduzierung des Kommunikationsaufwandes sollten darüber hinaus möglichst wenige Daten zwischen den Prozessoren ausgetauscht werden müssen, d. h., die Anzahl der Kanten, deren Endpunkte

in verschiedenen Partitionen liegen, sollte minimiert werden. Gegebenenfalls ist die Zeit, die für einen Datenaustausch benötigt wird, in die Ablaufplanung mit einzubeziehen.

- **Lokalitätsoptimierung**: Der durch das Scheduling-Verfahren festgelegte Ablaufplan bestimmt auch die Reihenfolge der Speicherzugriffe und somit das Lokalitätsverhalten. Bereits im vorigen Abschnitt, in dem der Einsatz von Techniken zur dynamischen Lastbalancierung diskutiert wurde, wurde darauf hingewiesen, daß beispielsweise eine Breitensuche auf dem Taskgraphen zu einem anderen Lokalitätsverhalten führt, als eine Tiefensuche. Plant man die Ausführung der Knoten einer Partition auf einem Prozessor, muß dazu eine topologische Reihenfolge der Knoten dieser Partition festgelegt werden. Diese sollte ein möglichst gutes Lokalitätsverhalten besitzen, damit in der Praxis eine möglichst geringe Laufzeit erzielt werden kann. Folgende Überlegungen sollten deshalb bei der Realisierung eines Scheduling-Verfahrens berücksichtigt werden:

  1. **Wiederverwendung des Ergebnisses der letzten Operation:** Eine mögliche Strategie zur Ausnutzung der zeitlichen Lokalität besteht darin, immer das Ergebnis der letzten Operation als Eingabe für die nächste Operation zu nutzen, indem nach der Ausführung eines Knotens immer als nächstes ein Kind dieses Knotens ausgeführt wird. Dies entspricht einer Tiefensuche.
  2. **Mehrfache Lese- oder Schreibzugriffe auf ein Datum:** Besitzt ein Knoten mehrere eingehende oder mehrere ausgehende Kanten, bedeutet dies, daß das Ergebnis des Knotens entweder mehrfach aktualisiert oder mehrfach gelesen wird. Diese zeitliche Lokalität kann ausgenutzt werden, indem die Vorfahren bzw. die Kinder dieses Knoten zeitlich nahe zur Ausführung des Knotens selbst geplant werden.
  3. **Speicherplatzbedarf:** Die Menge der Zwischenergebnisse, für die in einem Ausführungsschritt des Ablaufplans Speicherplatz bereit gestellt werden muß, wird dadurch bestimmt, wieviele Kanten existieren, deren Quellknoten vor und deren Zielknoten nach dem betrachteten Ausführungsschritt ausgeführt werden. Minimiert man über alle Ausführungsschritte die maximale Anzahl solcher Kanten, minimiert man auch den Speicherplatzbedarf und verbessert dadurch die Lokalität.
  4. **Räumliche Wiederverwendung von Cachezeilen:** Wird bei Ausführung eines Knotens auf ein Datum zugegriffen, das Element eines Vektors ist, kann man die räumliche Lokalität ausnutzen, indem man Operationen, die auf benachbarte Vektorelemente zugreifen, zeitlich nahe zu dieser Operation plant. Dazu ist es erforderlich, daß derartige Nachbarschaftsbeziehungen zwischen Vektorelementen auf geeignete weise modelliert werden.

Gegenüber optimierenden Compilern hat dieses Vorgehen den Vorteil, daß die Ablaufplanung anhand des Datenflusses erfolgt und deshalb zunächst keine Vorgaben bezüglich vorhandener Datenstrukturen und bezüglich des Kontrollflusses zwingend notwendig sind. Ein Compiler, der einen Quelltext in ein ausführbares Programm transformiert, muß dagegen den durch den Quelltext vorgegebenen Kontrollfluß mittels der ebenfalls vorgegebenen Datenstrukturen umsetzen. Gleichzeitig hat die Ablaufplanung anhand des Datenflußgraphen aber auch den Nachteil, daß vorab eine Festlegung auf eine bestimmte Problemgröße erfolgen muß und das erzeugte Programm eine sequentielle Instruktionsfolge ohne die Möglichkeit der Verwendung von Schleifen darstellt. Eine effiziente Nutzung des Instruktionscaches ist daher nicht möglich. Die Einschränkung auf eine feste Problemgröße ist problematisch, weil bei jeder Veränderung der Problemgröße eine neue Ablaufplanung erfolgen muß. Doch auch eine Optimierung auf Basis des Kontrollflusses müßte für jede neue Problemgröße wiederholt werden, um die für die jeweilige Zielarchitektur bestmögliche Laufzeit erreichen zu können, denn für unterschiedliche Problemgrößen können unterschiedliche Schleifenstrukturen oder unterschiedliche Blockgrößen optimal sein.

4.9 Zusammenfassung

In diesem Kapitel wurde der Ansatz verfolgt, das Lokalitätsverhalten und die Skalierbarkeit eingebetteter Runge-Kutta-Verfahren durch eine Spezialisierung bezüglich des Problems zu verbessern. Dazu wurde davon ausgegangen, daß die zu integrierenden Probleme eine beschränkte Zugriffsdistanz besitzen, d. h., daß ihre Zugriffsdistanz $d(\mathbf{f})$ klein im Verhältnis zur Systemgröße n ist. Alle Probleme aus dieser Klasse haben

die Eigenschaft, daß die Funktionsauswertung $f_j(t, \mathbf{w})$ für eine Komponente j nur auf Komponenten des Argumentvektors innerhalb eines kleinen, begrenzten Indexbereiches zugreift, der j enthält. Dies führt zu einer teilweisen Entkoppelung der Stufen, die neue Möglichkeiten zur Transformation des Programmcodes bietet.

Als erstes wurden die neuen Transformationsmöglichkeiten zur Optimierung der Speicherzugriffslokalität im sequentiellen Fall genutzt. Das Lokalitätsverhalten der allgemeinen Implementierungen ist durch die Wiederverwendung vollständiger Vektoren geprägt, da die Stufen aufgrund der Datenabhängigkeiten sequentiell berechnet werden müssen und in jeder Stufe über mehrere Vektoren iteriert wird. Unter der Voraussetzung einer beschränkten Zugriffsdistanz kann die Schleifenstruktur so verändert werden, daß die Schleife über die Systemdimension zur äußersten Schleife wird und in jedem Schritt dieser Schleife ein oder mehrere Elemente jedes Argumentvektors berechnet werden. Aufgrund der verbliebenen Abhängigkeiten, die eine vollständige Entkoppelung der Stufen verhindern, muß dabei die Berechnung der Argumentvektoren um mindestens je $d(\mathbf{f})$ Komponenten versetzt erfolgen, woraus ein Berechnungsschema ähnlich dem einer Pipeline resultiert, das einen diagonalen Lauf über die Argumentvektoren durchführt. Bei einer blockbasierten Implementierung unter Verwendung der Blockgröße B, wird dadurch die Größe des Arbeitsraumes R_I der äußeren Schleife über die Systemdimension reduziert auf $\Theta(s^2B)$. Als Folge davon entsteht im Vergleich zur allgemeinen Ausgangsimplementierung eine zusätzliche Arbeitsmenge, die unter der Voraussetzung $d(\mathbf{f}) \ll n$ wesentlich kleiner als ein vollständiger Vektor der Größe n ist und deshalb zu einer erheblichen Verringerung der Anzahl der Cache-Fehlzugriffe führt, sofern der Cache eine ausreichende Größe besitzt, um diese Arbeitsmenge aufnehmen zu können, er jedoch zu klein ist, um mehrere vollständige Vektoren speichern zu können.

Es wurden mehrere verschiedene sequentielle Varianten des Pipelining-Algorithmus implementiert, wobei für alle Pipelining-Implementierungen die allgemeine Implementierung (D) als Basisimplementierung verwendet wurde. Zur Untersuchung des Lokalitätsverhaltens dieser Implementierungen wurden zunächst Simulationsexperimente mit Hilfe des Simulators Simics durchgeführt, in denen zur Identifizierung der Arbeitsmengen die Zahl der Cache-Fehlzugriffe in Abhängigkeit von der Cachegröße betrachtet wurde. Die Ergebnisse der Simulationsexperimente bestätigen das Ausbilden einer zusätzlichen Arbeitsmenge, deren Größe etwa mit dem vorhergesagten Arbeitsraum eines Pipelining-Schrittes korrespondiert. Überschreitet die Cachegröße diese Arbeitsmenge, sinkt für die Pipelining-Implementierungen die Anzahl der Cache-Fehlzugriffe deutlich unter die Anzahl von Cache-Fehlzugriffen, die für Implementierung (D) beobachtet wurde. Für geringere Cachegrößen und ab einer Cachegröße, die in der Lage ist, mehrere vollständige Vektoren zu speichern, verhalten sich die blockbasierten Pipelining-Implementierungen ähnlich wie die Ausgangsimplementierung (D). Für eine elementweise arbeitende Pipelining-Implementierung ergab die Simulation dagegen für kleine Cachegrößen eine höhere Anzahl von Cache-Fehlzugriffen als für Implementierung (D).

Zur Untersuchung des Lokalitäts- und Laufzeitverhaltens auf realen Rechnersystemen wurden Laufzeitexperimente auf verschiedenen Zielplattformen durchgeführt. Auf einem Itanium-2-System wurde dazu für die Testprobleme MEDAKZO und BRUSS2D der Verlauf der bezüglich der Systemgröße und der Anzahl der Zeitschritte normierten Laufzeit in Abhängigkeit von der Systemgröße untersucht. Während für die allgemeinen Implementierungen (D) und (Dblock) ab einer bestimmten Systemgröße, ab der nicht mehr alle verwendeten Vektoren vollständig im Cache gespeichert werden können, die normierte Laufzeit stark ansteigt, erhöht sich die normierte Laufzeit der blockbasierten Pipelining-Implementierung (PipeD) an dieser Stelle nur geringfügig, so daß sie für große Systemgrößen eine erheblich bessere normierte Laufzeit als (D) und (Dblock) erzielt. Auf weiteren Zielplattformen wurden mehrere Einzelexperimente mit den Testproblemen MEDAKZO und BRUSS2D unter Verwendung unterschiedlicher Systemgrößen durchgeführt. Auch auf diesen Systemen kann für große Systemgrößen durch Verwendung des Pipelining-Ansatzes in der Regel eine Laufzeitverbesserung erzielt werden.

Da die Pipelining-Implementierungen in jedem Pipelining-Schritt eine oder mehrere Komponenten des Fehlervektors berechnen, kann in bestimmten Fällen frühzeitig erkannt werden, ob der gerade berechnete Zeitschritt verworfen werden muß. Dadurch ist es möglich, Zeitschritte, die verworfen werden müssen, frühzeitig abzubrechen und somit Rechenzeit einzusparen. Ein Nachteil dieses Vorgehens besteht jedoch darin, daß unter Umständen der lokale Fehler zu klein geschätzt wird, da nicht alle Fehlerkomponenten bekannt sind. Dem kann entgegengewirkt werden, indem beispielsweise in jedem Zeitschritt mindestens bis zu dem Index gerechnet wird, an dem im vorhergehenden Zeitschritt die Komponente des lokalen Feh-

lers den höchsten Wert hatte. In den durchgeführten Laufzeitexperimenten konnte durch dieses Vorgehen jedoch kein oder nur eine geringer Laufzeitgewinn erzielt werden, da die verwendete Schrittweitenkontrolle nur eine kleine Zahl verworfener Zeitschritte erzeugte und durch das Prüfen der Abbruchbedingung ein geringer zusätzlicher Overhead entsteht.

Bei der Integration sehr großer gewöhnlicher Differentialgleichungssysteme, wie sie z. B. durch die Ortsdiskretisierung von partiellen Differentialgleichungssystemen entstehen, kann der Speicherplatzbedarf für mehrstufige allgemeine Implementierungen inakzeptabel hoch sein. Von verschiedenen Autoren, u. a. Berland u. a. (2005) und Calvo u. a. (2004), wurden deshalb spezielle explizite Runge-Kutta-Verfahren vorgeschlagen, deren Verfahrensmatrix nicht voll besetzt ist, so daß eine Implementierung mit einem Speicherbedarf von z. B. $2n$ oder $3n$ möglich ist. Da die Verfahrensmatrix nicht voll besetzt sein kann und speziellen Anforderungen genügen muß, unterliegen auch die numerischen Eigenschaften der vorgeschlagenen Verfahren einer Reihe von Einschränkungen. Darüber hinaus enthalten die Verfahren häufig keine eingebetteten Lösungen für die Durchführung einer effizienten Schrittweitenkontrolle. Da zur Realisierung eines Pipelining-Algorithmus lediglich der Arbeitsraum eines Pipelining-Schrittes sowie ein Approximationsvektor gespeichert werden muß, kann dieser bei Verwendung einer Blockgröße B mit einem Speicherplatzbedarf von nur $n + O\left(s^2 B\right)$ realisiert werden. Die Verfahrensmatrix unterliegt dabei keinerlei Einschränkungen. Eine effiziente Schrittweitenkontrolle ist mit einem zusätzlichem Speicherplatzbedarf von n realisierbar. Ein Nachteil gegenüber den von anderen Autoren vorgeschlagenen $2n$-Verfahren besteht jedoch darin, daß der Pipelining-Ansatz nur auf Probleme mit beschränkter Zugriffsdistanz anwendbar ist.

Implementierungen für verteilten Adreßraum können eine beschränkte Zugriffsdistanz zusätzlich zur Lokalitätsoptimierung dazu ausnutzen, die Interprozessorkommunikation effizienter zu organisieren. Da aufgrund der beschränkten Zugriffsdistanz während der Berechnung der Stufen Kommunikation nur noch zwischen benachbarten Prozessoren erforderlich ist, können Einzeltransferoperationen zur Realisierung des Datenaustauschs genutzt werden, während in einer allgemeinen Implementierung in jeder Stufe eine Multibroadcastoperation ausgeführt werden muß. Es wurden verschiedene Implementierungsvarianten vorgestellt, die mit Hilfe von Einzeltransferoperationen kommunizieren und dabei versuchen, die Datenübertragungszeiten durch Berechnungen zu überdecken. Diese Implementierungsvarianten berechnen die Stufen entweder nacheinander, oder sie nutzen das Pipelining-Berechnungsschema zusammen mit verschiedenen Finalisierungsvarianten zur Verbesserung des Lokalitätsverhaltens. Eine Untersuchung des Skalierbarkeitsverhaltens dieser spezialisierten Implementierungen auf verschiedenen Zielsystemen mit unterschiedlicher Architektur hat gezeigt, daß diese in der Lage sind, erheblich höhere Speedups als die allgemeinen Implementierungen zu erzielen. Die Laufzeit wird dabei wesentlich durch den Kommunikationsaufwand bestimmt. Es konnten jedoch auch signifikante Auswirkungen des Lokalitätsverhaltens auf die parallele Laufzeit beobachtet werden, insbesondere auf Parallelrechnern mit gemeinsamem Adreßraum.

Bei der Realisierung spezialisierter Implementierungen für gemeinsamen Adreßraum können die beteiligten Threads die verwendeten Vektoren gemeinsam nutzen. Ein expliziter Datenaustausch ist daher nicht erforderlich, es muß jedoch eine Synchronisation der Threads mit Hilfe von Barrier-Operationen oder Mutex-Variablen durchgeführt werden. Die Berechnungsreihenfolge, wie sie zur Realisierung der Implementierungen für verteilten Adreßraum verwendet wurde, kann dabei beibehalten werden. Dieser Vorgehensweise folgend, wurden mehrere Implementierungsvarianten für gemeinsamen Adreßraum erstellt, die jeweils die gleiche Berechnungsreihenfolge wie eine korrespondierende Implementierung für verteilten Adreßraum nutzen, jedoch anstelle des Nachrichtenaustausches geeignete Mechanismen zur Threadsynchronisation verwenden.

Für eine dieser Implementierungsvarianten, die das Pipelining-Berechnungsschema umsetzt, wurde analog zur Untersuchung der sequentiellen Implementierungsvarianten das Verhalten der normierten Laufzeit in Abhängigkeit von der Systemgröße auf einem mit 4 Prozessoren ausgestatteten Itanium-2-System untersucht. Dabei bestätigte sich, daß die Pipelining-Implementierung durch die parallele Ausführung für große Systemgrößen aufgrund des kleineren Arbeitsraumes ihrer äußeren Schleife eine deutlich stärkere Beschleunigung erfährt als die zugehörige allgemeine Basisimplementierung. Für kleine Systemgrößen, die es ermöglichen, alle verwendeten Vektoren im Cache vorzuhalten, ist der Laufzeitunterschied zwischen beiden Implementierungen dagegen nur gering. Die Systemgröße, ab der die Pipelining-Implementierung eine deutlich geringere Laufzeit erreicht als die allgemeine Basisimplementierung, erhöht sich mit steigender Prozessorzahl, da sich die Arbeitsmengen der allgemeinen Implementierung durch die Aufteilung auf mehrere Prozessoren verringern.

Eine Skalierbarkeitsuntersuchung der erstellten Implementierungen auf zwei weiteren Zielplattformen mit größerer Prozessorzahl führte ebenfalls zu dem Ergebnis, daß die spezialisierten Implementierungen in der Lage sind, höhere Speedups als die allgemeinen Implementierungen zu erzielen. Allerdings wurden weniger große Laufzeitunterschiede im Vergleich zu den allgemeinen Implementierungen gemessen als für die Implementierungen für verteilten Adreßraum, da die allgemeinen Implementierungen für gemeinsamen Adreßraum höhere Speedups erreichen als die allgemeinen Implementierungen für verteilten Adreßraum. Außerdem erreichten die spezialisierten Implementierungen für verteilten Adreßraum teilweise höhere Speedups als die spezialisierten Implementierungen für gemeinsamen Adreßraum, was darauf hindeutet, daß die Implementierungen für gemeinsamen Adreßraum durch die gemeinsame Nutzung der Vektoren durch alle beteiligten Threads die NUMA-Architektur der verwendeten Zielsysteme nicht optimal ausnutzen. Es ist jedoch davon auszugehen, daß bei Verwendung verteilter Datenstrukturen in Kombination mit einem expliziten, durch das Kopieren von Speicherbereichen realisierten Datenaustausch mindestens ebenso gute Speedups wie für die MPI-Implementierungen erreicht werden könnten.

Daß das Lokalitätsverhalten einen wichtigen Einfluß auf die Skalierbarkeit ausübt, kann anhand des Verhaltens der spezialisierten Implementierung (Dbc) festgestellt werden. Das Lokalitätsverhalten dieser Implementierung ist nahezu identisch mit dem der allgemeinen Implementierung (D), bezüglich der verwendeten Synchronisationsmechanismen entspricht sie jedoch weitestgehend der Pipelining-Implementierung (PipeD). In vielen Experimenten bestimmt das Lokalitätsverhalten die Laufzeit, weshalb die Laufzeit von (Dbc) etwa mit der Laufzeit von (D) übereinstimmt, während die Pipelining-Implementierung (PipeD) eine bessere Laufzeit erzielen kann. Doch auch die Synchronisationsmechanismen können laufzeitbestimmend wirken, insbesondere bei der Integration von Problemen mit relativ geringer Systemgröße bei Verwendung einer hohen Prozessoranzahl. In diesen Fällen arbeitet (Dbc) deutlich schneller als (D) und erreicht oder übertrifft eventuell sogar Implementierung (PipeD).

Zur Ausnutzung der spezifischen Architektur von SMP-Clustersystemen können hybride Implementierungen genutzt werden, die verschiedene Programmiermodelle, z. B. MPI und POSIX Threads, kombinieren. Solche Implementierungen wurden im Rahmen einer Bachelor-Arbeit (Behrend 2005) realisiert und untersucht. Eine wesentliche Laufzeitverbesserung gegenüber reinen MPI-Implementierungen konnte mit den betrachteten Implementierungen jedoch nicht erzielt werden. Als eine der möglichen Ursachen wurde ein Lastungleichgewicht identifiziert, das dadurch hervorgerufen wird, daß Threads, die MPI-Operationen ausführen, eine höhere Laufzeit benötigen als andere Threads.

Ein Ausgleich dieses Lastungleichgewichts könnte durch die Verwendung von Lastbalancierungstechniken erreicht werden. Der Einsatz verschiedener Lastbalancierungsstrategien wurde bereits im Zusammenhang mit allgemeinen parallelen Implementierungen untersucht. In den dabei durchgeführten Experimenten auf Rechnersystemen mit gemeinsamem Adreßraum konnte teilweise sogar für Probleme mit über die einzelnen Komponenten nahezu gleichverteilten Funktionsauswertungskosten eine zumindest geringe Laufzeitverbesserung erzielt werden. Der Einsatz von Lastbalancierungsverfahren erscheint auch für spezialisierte Implementierungen äußerst vielversprechend, da hier durch die weitgehende Entkoppelung der Stufen die Berechnung der einzelnen Stufen nicht durch Barrier-Operationen getrennt werden muß. Insbesondere für hybride Implementierungsvarianten ist durch den Einsatz von Lastbalancierungstechniken eine signifikante Verbesserung der Laufzeit zu erwarten, da ohne Lastbalancierung Threads, die keine MPI-Operationen ausführen, auf Threads, die mittels MPI mit anderen Knoten kommunizieren, warten müssen. Allerdings müssen bei der dynamischen Ablaufplanung die trotz der beschränkten Zugriffsdistanz verbliebenen Abhängigkeiten der Funktionsauswertung berücksichtigt werden.

Die in diesem Abschnitt vorgestellten Techniken zur Optimierung des Lokalitätsverhaltens und zur Reduzierung des Kommunikations- bzw. Synchronisationsaufwandes durch Ausnutzung der beschränkten Zugriffsdistanz können auch auf andere Lösungsverfahren übertragen werden. Als Beispiel wurden Möglichkeiten zur Lokalitätsoptimierung sequentieller iterierter Runge-Kutta-Verfahren im Rahmen einer Bachelor-Arbeit (Gerkhardt 2005) untersucht. Hierbei bestätigte sich, daß auch für iterierte Runge-Kutta-Verfahren durch Ausnutzung der beschränkten Zugriffsdistanz eine Reduzierung der Anzahl der Cache-Fehlzugriffe und daraus resultierend eine Laufzeitverbesserung erzielt werden kann.

5 Zusammenfassung und Ausblick

Das Thema dieser Arbeit ist die effiziente Implementierung eingebetteter Runge-Kutta-Verfahren zur Lösung von Anfangswertproblemen gewöhnlicher Differentialgleichungssysteme auf modernen sequentiellen und parallelen Rechnersystemen. Als wichtigster Ansatz wurde versucht, die Laufzeit durch Ausnutzung der Speicherzugriffslokalität zu reduzieren. Eine Verbesserung der Speicherzugriffslokalität verringert die Anzahl der Cache-Fehlzugriffe und führt infolgedessen in der Regel zu einer Laufzeitverbesserung, da weniger häufig Zugriffe auf den Hauptspeicher ausgeführt werden müssen, die im Vergleich zu Cache-Zugriffen vielfach höhere Zugriffszeiten benötigen. Von einer Verringerung der Anzahl der Cache-Fehlzugriffe profitieren sowohl sequentielle als auch parallele Implementierungen. Insbesondere auf Parallelrechnern mit gemeinsamem Adreßraum kann das Lokalitätsverhalten entscheidend für die erreichbare Skalierbarkeit sein, da das interne Verbindungsnetzwerk, das Prozessoren und Speichermodule untereinander verbindet, nur eine begrenzte Kapazität besitzt. Neben der Speicherzugriffslokalität hat jedoch auch die Kommunikation zwischen den beteiligten Kontrollflüssen einen entscheidenden Einfluß auf die Skalierbarkeit. Dies gilt insbesondere für parallele Rechnersysteme mit verteiltem Adreßraum, deren Prozessoren durch Nachrichtenaustausch kommunizieren müssen. Als zweiter Ansatz zur Erhöhung der parallelen Effizienz wurde deshalb versucht, den Kommunikationsaufwand sowie durch die Synchronisation von Kontrollflüssen entstehende Wartezeiten zu reduzieren.

5.1 Überblick über sequentielle und parallele Lösungsverfahren

Zu Beginn der Arbeit wurde die betrachtete Problemklasse der Anfangswertprobleme gewöhnlicher Differentialgleichungssysteme vorgestellt, und es wurde ein Überblick über existierende numerische Verfahren zu deren Lösung gegeben. Dazu wurden explizite und implizite Runge-Kutta-Verfahren, Extrapolationsverfahren, lineare Mehrschrittverfahren, allgemeine lineare Methoden und Waveform-Relaxationsverfahren sowie Möglichkeiten zur Schrittweiten- und Ordnungssteuerung vorgestellt, und es wurden ausführliche Literaturhinweise gegeben. Eingebettete Runge-Kutta-Verfahren ordnen sich in diese Verfahrensklassen als spezielle explizite Runge-Kutta-Verfahren ein, die durch die Bereitstellung einer oder mehrerer eingebetteter Lösungen eine effiziente Schrittweitenkontrolle ermöglichen. Sie werden häufig in der Praxis eingesetzt, da sie aufgrund ihrer numerischen Eigenschaften sehr gut für die Lösung nichtsteifer Anfangswertprobleme geeignet sind, gleichzeitig aber aufgrund ihrer im Vergleich zu anderen Verfahren einfacheren Berechnungsvorschrift sehr effizient arbeiten und dennoch einfach zu realisieren sind.

Bei der Parallelisierung von Lösungsverfahren für gewöhnliche Differentialgleichungssysteme unterscheidet man aus Sicht des Lösungsverfahrens folgende Parallelitätsarten:

- Parallelität bezüglich des Systems,
- Parallelität bezüglich der Methode und
- Parallelität bezüglich der Zeitschritte.

Aus algorithmischer Sicht unterscheidet man zwischen:

- Instruktionsparallelität,
- Datenparallelität und
- Taskparallelität.

Es wurde ein Überblick darüber gegeben, inwiefern explizite und implizite Runge-Kutta-Verfahren, iterierte Runge-Kutta-Verfahren, Extrapolationsverfahren, Mehrschrittverfahren und Waveform-Relaxationsverfahren diese Parallelitätsarten aufweisen und für eine parallele Implementierung geeignet sind. Eingebettete Runge-Kutta-Verfahren besitzen im allgemeinen Fall aufgrund der Abhängigkeiten zwischen den Stufen und zwischen aufeinanderfolgenden Zeitschritten nur ein geringes Parallelitätspotential bezüglich der Methode und können nicht mehrere Zeitschritte parallel ausführen. Sie besitzen jedoch ein großes Parallelitätspotential bezüglich des Systems, da in jeder Stufe eine unabhängige Berechnung der einzelnen Systemkomponenten möglich ist.

5.2 Programmtechnische Realisierung eingebetteter Runge-Kutta-Verfahren

Bei der programmtechnischen Realisierung eingebetteter Runge-Kutta-Verfahren sind eine Reihe von Entwurfsaspekten zu berücksichtigen. Die Möglichkeiten, die dabei gegeben sind, werden zuallererst durch die Wahl der Programmiersprache bestimmt. Alle in dieser Arbeit betrachteten Implementierung verwenden die Programmiersprache C, da diese einerseits portabel ist und somit die Untersuchung der erstellten Implementierungen auf einer Vielzahl unterschiedlicher Plattformen ermöglicht, sie andererseits jedoch sehr maschinennah ist, so daß der Programmierer sowohl den Kontroll- als auch den Datenfluß weitestgehend selbst vorgeben kann und ihm somit die Möglichkeit zur Optimierung des Programmcodes und zur Ausnutzung architekturspezifischer Eigenschaften der betrachteten Zielplattformen gegeben ist.

Entwurfsaspekte, die bei der Implementierung eingebetteter Runge-Kutta-Verfahren berücksichtigt werden müssen, sind u. a.:

- Wie soll die rechte Seite des gewöhnlichen Differentialgleichungssystems realisiert werden? Soll das Verfahren nur für ein einziges Problem implementiert werden, oder soll es durch Austauschbarkeit der Funktion $\mathbf{f}$ auf verschiedene Probleme angewendet werden können? Soll die Programmfunktion zur Realisierung der rechten Seite als Vektorfunktion $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ oder als skalare Funktion $f : \mathbb{N} \times \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}$ realisiert werden?
- Welche Verfahrenskoeffizienten sollen verwendet werden? Liegen diese fest oder sollen verschiedene Verfahrenskoeffizienten verwendet werden können? Ist die Anzahl der Stufen festgelegt?
- Wird eine stetige Erweiterung benötigt? Welche Abbruchkriterien soll es geben und auf welche Ereignisse soll reagiert werden?
- Welche statistischen Informationen über den Integrationsvorgang werden benötigt?
- Wie soll die Schnittstelle zum Anwendungsprogramm realisiert werden?

Es existieren eine Reihe von Softwareimplementierungen eingebetteter Runge-Kutta-Verfahren, u. a. DOPRI5 und DOP853, DVERK, RKSUITE sowie die MATLAB-ODE-Suite, die z. T. in mathematische Bibliotheken und Softwarepakete eingeflossen sind. Beim Entwurf dieser Implementierungen standen vor allem die numerischen Eigenschaften sowie die Bereitstellung einer bestimmten Funktionalität im Vordergrund. Laufzeitoptimierungen wurden hauptsächlich unter numerischen Gesichtspunkten durchgeführt.

Da das Ziel dieser Arbeit darin besteht, durch Optimierung der Speicherzugriffslokalität und Reduzierung des Synchronisationsoverheads eine Laufzeitverbesserung herbeizuführen, wurden u. a. folgende Entwurfsentscheidungen getroffen:

- Es sollen beliebige Verfahrenskoeffizienten verwendet werden können, und auch die Stufenzahl soll keiner Einschränkung unterliegen.
- Die rechte Seite des gewöhnlichen Differentialgleichungssystems wird als skalare, externe Funktion $f : \mathbb{N} \times \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}$ realisiert, um Schleifentransformationen, eine Parallelisierung bezüglich des Systems sowie die Austauschbarkeit des Problems zu ermöglichen.

- Es wird nur der Grundalgorithmus realisiert. Auf zusätzliche Funktionalität wird verzichtet. Der gesamte Grundalgorithmus soll möglichst innerhalb einer einzigen Programmfunktion implementiert werden, um zusätzliche Kosten aufgrund von Funktionsaufrufen zu vermeiden.

Diesen Entscheidungen entsprechend, wurde das in Abb. 2.1 dargestellte Rahmenprogramm entwickelt, an dem sich alle im Verlauf dieser Arbeit vorgestellten Implementierungen orientieren. Verschiedene Implementierungsvarianten unterscheiden sich nur anhand des Berechnungskernels, der die Berechnung der Stufen- und Argumentvektoren sowie der Hilfsvektoren steuert.

5.3 Ansätze zur Laufzeitverbesserung

Um ein gegebenes Anfangswertproblem in möglichst kurzer Zeit zu lösen, ist es zunächst wichtig, eine geeignete Plattform mit schnellen Prozessoren, effizienter Speicherhierarchie, hohem Parallelitätsgrad und gegebenenfalls schnellem Verbindungsnetzwerk auszuwählen, auf der effiziente Compiler verfügbar sind. Um diese Plattform bestmöglich zu nutzen, muß der Programmcode so gestaltet werden, daß

- die Anzahl der sequentiell ausgeführten Instruktionen möglichst gering ist,
- die Anzahl der Wartezyklen, die benötigt werden, um Operationen oder Operanden zu laden, möglichst gering ist und
- die aktive oder passive Wartezeit aufgrund von Kommunikation bzw. Synchronisation mit anderen Kontrollflüssen möglichst gering ist.

Bei der Implementierung können zur Einsparung von Instruktionen verschiedene Techniken angewendet werden, z. B. die Abspaltung von Schleifeniterationen, die Minimierung der Anzahl der Multiplikationen mit der Schrittweite durch Ausnutzung der Distributivität, die Reduzierung der Dimension von Feldzugriffen und das Verschmelzen von Schleifen.

Lokalitätsoptimierungen zur Reduzierung der Anzahl der Wartezyklen für Speicherzugriffe auf den Programmcode sind in vielen Fällen nicht erforderlich, da der Programmcode nur wenige Sprünge zur Realisierung kleiner, kompakter Schleifen enthält und die Größe des Programmcodes eines Zeitschrittes auf vielen Systemen die Größe des Instruktionscaches nicht überschreitet.

Die Lokalität der Datenzugriffe wird u. a. durch die während der Auswertung der Funktion $\mathbf{f}$ ausgeführten Speicherzugriffe beeinflußt. Als Maß für die Lokalität der Funktionsauswertung kann die *Zugriffsdistanz* (Definition 2.2) der Funktion $\mathbf{f}$, $d(\mathbf{f})$, verwendet werden.

Weiterhin hat die Auswahl und Organisation der Datenstrukturen einen wichtigen Einfluß auf das Lokalitätsverhalten. Den größten Speicherplatzbedarf und gleichzeitig den größten Einfluß auf das Lokalitätsverhalten besitzen die Stufenvektoren $\mathbf{v}_1, \ldots, \mathbf{v}_s$, die Argumentvektoren $\mathbf{w}_1, \ldots, \mathbf{w}_s$, der Approximationsvektor η sowie die Hilfsvektoren $\Delta\eta$, $\mathbf{e}$ und $\mathbf{s}$, da deren Größe der Dimension des zu integrierenden gewöhnlichen Differentialgleichungssystems entspricht. Für Systeme mit hoher Dimension ist es in der Regel nicht möglich, alle Vektoren im Cache zu speichern.

Die Speicherzugriffe auf diese Datenstrukturen erfolgen innerhalb einer Schleifenstruktur, die aus der Umsetzung der Berechnungsvorschrift eines Zeitschrittes resultiert. Es bestehen abhängig von der Auswahl der Datenstrukturen verschiedene Möglichkeiten, diese Schleifenstruktur zu realisieren. Zum Beispiel ist es bei Speicherung aller Stufenvektoren möglich, die Berechnung der Hilfsvektoren gemeinsam mit der Berechnung der Stufenvektoren durchzuführen oder diese Berechnungen voneinander zu trennen. Die Realisierung der Schleifenstruktur bestimmt die Reihenfolge und die Anzahl der Speicherzugriffe auf die verwendeten Vektoren und ist damit ausschlaggebend für das Lokalitätsverhalten.

Durch eine Spezialisierung auf feste Verfahrenskoeffizienten oder eine feste Stufenanzahl wären zusätzliche Optimierungen möglich, wie z. B. das Aufrollen von Schleifen.

5.4 Effiziente Implementierung für Gleichungen mit beliebigen Abhängigkeiten

5.4.1 Untersuchte Schleifenstrukturen

Bei der Realisierung der Schleifenstruktur zur Umsetzung der Berechnungsvorschrift eines Zeitschrittes wird die Anzahl der möglichen Schleifenstrukturen dadurch eingeschränkt, daß die Funktionsauswertung einer Komponente, $f_j(t, \mathbf{w})$, im allgemeinen Fall auf alle Komponenten des Argumentvektors $\mathbf{w}$ zugreifen kann. Die Stufenvektoren $\mathbf{v}_1, \ldots, \mathbf{v}_s$ bzw. die Argumentvektoren $\mathbf{w}_1, \ldots, \mathbf{w}_s$ müssen deshalb nacheinander berechnet werden. Die äußerste Schleife der Stufenberechnung läuft deshalb immer über die Stufen.

Beispielhaft wurden vier Implementierungsvarianten vorgestellt, die auf unterschiedlichen Schleifenstrukturen basieren. Jede dieser Implementierungsvarianten zeichnet sich dadurch aus, daß sie versucht, eine bestimmte Art der Lokalität möglichst optimal auszunutzen.

- **Implementierung (A)** verfolgt das Ziel einer möglichst guten Ausnutzung der räumlichen Lokalität. Sie arbeitet deshalb vektororientiert, d. h., alle Vektoren werden elementweise durchlaufen, indem die innersten Schleifen über die Systemdimension iterieren. Es werden alle Stufenvektoren gespeichert. Die Argumentvektoren werden in einem temporären Vektor $\mathbf{w}$ gespeichert, der in jeder Stufe mit dem Wert des aktuellen Argumentvektors überschrieben wird.

- **Implementierung (E)** verfolgt das Ziel der Optimierung der zeitlichen Lokalität bezüglich der Schreibzugriffe auf Argumentvektorelemente. Sie benutzt die gleichen Datenstrukturen wie Implementierung (A). Die Schleifenstruktur wurde jedoch durch das Vertauschen und Verschmelzen von Schleifen so verändert, daß die innerste Schleife über die bereits berechneten Stufen iteriert, wobei jede Iteration der innersten Schleife einen Schreibzugriff auf das gleiche Argumentvektorelement ausführt.

- **Implementierung (D)** hat das Ziel der Optimierung der zeitlichen Lokalität bezüglich der Lesezugriffe auf Stufenvektorelemente. Dazu werden alle Argumentvektoren $\mathbf{w}_1, \ldots, \mathbf{w}_s$ im Speicher gehalten. Die innerste Schleife iteriert über die noch nicht fertig berechneten Stufen. Dabei verwendet sie ein durch eine Funktionsauswertung berechnetes Stufenvektorelement, um alle von diesem Stufenvektorelement abhängigen Argumentvektorelemente zu aktualisieren. Da zur Speicherung dieses Stufenvektorelements eine skalare Variable genügt, müssen die Stufenvektoren $\mathbf{v}_1, \ldots, \mathbf{v}_s$ nicht als Vektoren gespeichert werden.

- **Implementierung (Dblock)** basiert auf der gleichen Schleifenstruktur wie Implementierung (D). Sie versucht jedoch, durch ein Tiling der Schleifen über die Systemdimension sowohl eine gute Ausnutzung der zeitlichen als auch der räumlichen Lokalität zu erreichen. Dazu kann sie im wesentlichen die gleichen Datenstrukturen wie Implementierung (D) verwenden, muß jedoch zusätzlich einen temporären Vektor $\hat{\mathbf{v}}$ bereitstellen, der B Stufenvektorelemente zwischenspeichern kann. B bezeichnet in diesem Zusammenhang die für das Schleifen-Tiling verwendete Blockgröße.

5.4.2 Parallele Implementierung

Da eingebettete Runge-Kutta-Verfahren nur ein geringes Parallelitätspotential bezüglich der Methode, jedoch ein sehr hohes Parallelitätspotential bezüglich des Systems besitzen, wurde ausschließlich die Parallelisierung bezüglich des Systems durch Aufteilung der Iterationen der Schleifen über die Systemdimension auf die beteiligten Kontrollflüsse betrachtet. Die Aufteilung wurde dabei blockweise durchgeführt, da dies zu einer höheren Lokalität führt und im Fall eines verteilten Adreßraumes eine effizientere Kommunikation ermöglicht. Aufgrund der durch die Funktionsauswertung gegebenen Abhängigkeiten muß bei der Implementierung für verteilten Adreßraum in jeder Stufe der jeweilige Argumentvektor mittels einer Multibroadcastoperation (z. B. `MPI_Allgather()` bei Verwendung von MPI) ausgetauscht werden. Bei der Implementierung für gemeinsamen Adreßraum kann man auf gleiche Weise vorgehen, indem man verteilte Datenstrukturen nutzt und eine Multibroadcastoperation durch das Kopieren der entsprechenden Daten emuliert. In dieser Arbeit wurden ausschließlich Implementierungsvarianten für gemeinsamen Adreßraum untersucht, die gemeinsame Datenstrukturen nutzen. Zur Synchronisation der beteiligten Threads genügt

es in diesem Fall, in jeder Stufe eine bzw. zwei Barrier-Operationen auszuführen, um sicherzustellen, daß Funktionsauswertungen erst nach Fertigstellung der Berechnung des benötigten Argumentvektors ausgeführt werden und daß ein gegebenenfalls verwendeter temporärer Argumentvektor nicht überschrieben wird, bevor alle auf ihn zugreifenden Funktionsauswertungen abgeschlossen sind.

5.4.3 Theoretische und experimentelle Analyse des Lokalitätsverhaltens

Zur Untersuchung des Lokalitätsverhaltens der Implementierungsvarianten wurde zunächst eine theoretische Analyse der Arbeitsräume der in den Implementierungen verwendeten Schleifen sowie eine Abschätzung der daraus resultierenden Arbeitsmengen durchgeführt. Diese zeigen, daß alle Implementierungen Arbeitsmengen besitzen, die durch das Zugriffsverhalten der Funktionsauswertung bestimmt werden. Ausschlaggebend für das Lokalitätsverhalten ist jedoch die zeitliche Wiederverwendung von Vektoren zwischen aufeinanderfolgenden Stufen. Dies hat zur Folge, daß sich das Lokalitätsverhalten durch eine Parallelisierung verbessert, da jeder Prozessor nur den ihm durch die blockweise Verteilung zugeordneten Anteil der Vektoren verarbeitet. Durch diese Verbesserung des Lokalitätsverhaltens sind theoretisch superlineare Speedups möglich. Für Schleifen über die Systemdimension, in denen Funktionsauswertungen ausgeführt werden, hängt die Verkleinerung des Arbeitsraumes jedoch von der Zugriffsdistanz der Funktion **f** ab. Dadurch verbessert sich das Lokalitätsverhalten der Implementierungen (D) und (Dblock) für Probleme mit großer Zugriffsdistanz bei einer Parallelisierung nicht in gleichem Maße wie das der Implementierungen (A) und (E).

Zur Identifizierung der tatsächlich auftretenden Arbeitsmengen sowie zur Untersuchung der Auswirkung der Assoziativität und der Zeilengröße des Caches auf das Lokalitätsverhalten der sequentiellen Implementierungen wurden Simulationsexperimente mit Hilfe des Simulators Simics durchgeführt. Dabei wurden verschiedene Arten von Caches mit unterschiedlicher Kapazität, Zeilengröße und Assoziativität nachgebildet und die Anzahl der jeweils auftretenden Cache-Fehlzugriffe ermittelt. Die Simulationsergebnisse zeigen, daß alle Implementierungen von einer höheren Assoziativität profitieren. Sofern keine ungünstige Anordnung der Datenstrukturen vorliegt, führt eine höhere Assoziativität als 4 jedoch nur noch zu einer geringfügigen Verbesserung. Prinzipiell profitieren darüber hinaus alle Implementierungen von einer Erhöhung der Cachezeilengröße, da sie durch die elementweise Iteration über Vektoren eine hohe räumliche Lokalität aufweisen. Allerdings führt die Erhöhung der Zeilengröße auch zu einer Vergrößerung der auftretenden Arbeitsmengen, was für bestimmte Cachegrößen zu einer Verschlechterung der Fehlzugriffszahl führt. Die beobachteten Arbeitsmengen decken sich im wesentlichen mit den aus der theoretischen Analyse entstandenen Erwartungen. Insbesondere bestätigt sich, daß vor allem die zeitliche Wiederverwendung vollständiger Vektoren das Lokalitätsverhalten bestimmt, während andere Arbeitsmengen in der Regel wenig signifikant sind.

Zur Analyse der auf realen Rechnersystemen auftretenden Arbeitsmengen sowie zur Untersuchung der Auswirkungen der Arbeitsmengen auf die Laufzeit wurden Laufzeitexperimente auf einem Itanium-2-SMP-System mit einer dreistufigen Cache-Hierarchie durchgeführt, in denen die Laufzeit pro Systemkomponente und pro Zeitschritt in Abhängigkeit von der Systemgröße bestimmt wurde. Die Erhöhung dieser normierten Laufzeit ab einer bestimmten Systemgröße zeigt das Herausfallen einer Arbeitsmenge aus einer der Cache-Stufen an. Die auf diese Weise für die sequentiellen Implementierungsvarianten ermittelten Laufzeitkurven werden überwiegend durch das Herausfallen von Vektoren aus dem L3-Cache geprägt. Jedoch hat auch das Zugriffsverhalten des betrachteten Testproblems einen entscheidenden Einfluß. Für kleine Problemgrößen, die die Speicherung von mindestens einem oder zwei vollständigen Vektoren im L3-Cache gestatten, erreicht die vektororientierte Implementierung (A) durch ihre gute Ausnutzung der räumlichen Lokalität die beste Laufzeit. Größere Probleme können mit anderen Implementierungen effizienter integriert werden. Für das Testproblem BRUSS2D liefert in diesem Fall Implementierung (Dblock) die geringste Laufzeit. Für das Testproblem MEDAKZO mit einer sehr geringen Zugriffsdistanz ist dagegen Implementierung (E) erfolgreicher. Die parallele Ausführung der Implementierungen für gemeinsamen Adreßraum führt für alle untersuchten Testprobleme zu einer Laufzeitverbesserung. Dabei bestätigen die gemessenen normierten Laufzeiten die erwartete Verkleinerung der Arbeitsmengen. Die erreichten Speedups sind deshalb abhängig von der Systemgröße. Durch die Verkleinerung der Arbeitsmengen verbessern sich besonders stark solche Implementierungen wie (A) und (E), die sehr große Arbeitsmengen besitzen.

5.4.4 Theoretische Analyse der Skalierbarkeit

Für die parallelen Implementierungen für verteilten Adreßraum wurde eine theoretische Analyse der Skalierbarkeit durchgeführt. Diese Implementierungen nutzen explizite Kommunikationsoperationen, deren Laufzeit durch geeignete Laufzeitformeln beschrieben werden kann.

Betrachtet man die asymptotische Entwicklung der Laufzeit, so verringert sich mit wachsender Prozessorzahl bei konstanter Problemgröße die pro Prozessor notwendige Rechenzeit, jedoch erhöht sich in der Regel der Kommunikationsaufwand, falls die Topologie des Verbindungsnetzwerks kein vollständiger Graph ist. Die in jeder Stufe ausgeführte Allgather-Operation[1] hat dabei asymptotisch einen weniger großen Einfluß als die innerhalb der Schrittweitenkontrolle ausgeführte Allreduce-Operation, da die Nachrichtengröße für die Allgather-Operationen mit wachsender Prozessorzahl geringer wird, während die Nachrichtengröße für die Allreduce-Operation konstant bleibt. Steigt die Laufzeit der Allreduce-Operation mit wachsender Prozessorzahl an, wie dies für die meisten typischen Verbindungsnetzwerke der Fall ist, strebt deshalb der Speedup asymptotisch gegen 0, was auf ein schlechtes Skalierbarkeitsverhalten hinweist. Die Skalierbarkeit kann jedoch durch Erhöhung der Problemgröße verbessert werden, insbesondere dann, wenn die Kosten für die Funktionsauswertung einer Komponente des Differentialgleichungssystems mit steigender Problemgröße zunehmen. Auch im Fall, daß Prozessoranzahl und Problemgröße in gleichem Verhältnis gesteigert werden, ergibt sich eine gute Skalierbarkeit. Dies entspricht der Situation, daß versucht wird, den aus einer Erhöhung der Problemgröße resultierenden zusätzlichen Parallelitätsgrad durch Verwendung zusätzlicher Prozessoren auszunutzen. Besonders hoch ist die Skalierbarkeit in diesem Fall dann, wenn die Kommunikationsoperationen asymptotisch nicht teurer sind als die Funktionsauswertungskosten.

5.4.5 Laufzeit und Skalierbarkeit auf verschiedenen Rechnersystemen

Laufzeitexperimente zur Untersuchung des Laufzeitverhaltens der sequentiellen Implementierungsvarianten auf vier unterschiedlichen Zielplattformen auf Basis der Prozessoren Sun UltraSPARC III Cu, Intel Pentium 4, IBM Power4+ und AMD Opteron haben bestätigt, daß das Lokalitätsverhalten einen entscheidenden Einfluß auf die Laufzeit ausübt. Beispielsweise konnte für das Testproblem BRUSS2D-MIX bei Verwendung der Problemgröße $N = 1000$ auf einem UltraSPARC-III-Prozessor die Laufzeit durch Verwendung der Implementierung (Dblock) im Vergleich zu Implementierung (D) um über 38 % von 2 h 47 min auf 1 h 43 min verbessert werden. Die Optimierung des Lokalitätsverhaltens ist deshalb eine wichtige Voraussetzung für eine effiziente Ausnutzung moderner Prozessoren. Aufgrund unterschiedlicher Eigenschaften der Zielplattformen und des Zugriffsverhaltens der zu integrierenden Probleme sowie aufgrund der Tatsache, daß verschiedene Compiler bei der Übersetzung eines Programms oft unterschiedliche Programmtransformationen durchführen, kann jedoch keine Implementierungsvariante unter allen Umständen die beste Laufzeit liefern. Um eine bestmögliche Ausnutzung zu erreichen, ist deshalb für jede Zielplattform und für jedes zu integrierende Problem eine erneute Evaluierung und Optimierung erforderlich.

Zur Untersuchung des Skalierbarkeitsverhaltens der parallelen Implementierungen wurde ebenfalls eine Reihe von Laufzeitexperimenten auf verschiedenen Zielplattformen durchgeführt. Für die mit Hilfe von MPI realisierten Implementierungen für verteilten Adreßraum konnte bei diesen Experimenten keine gute Skalierbarkeit beobachtet werden. Vor allem auf Clustersystemen, die aus Standardkomponenten aufgebaut wurden und über Infiniband oder Gigabit bzw. Fast Ethernet verbunden waren, konnten nur sehr schlechte Speedup-Werte erzielt werden. Aber auch auf Parallelrechnern mit gemeinsamem Adreßraum erreichten die MPI-Implementierungen keine guten Speedup-Werte. Zum Beispiel konnte auf einem mit 32 Prozessoren ausgestatteten Knoten des Supercomputers JUMP nicht einmal ein Speedup von 4 erreicht werden. Eine bessere Skalierbarkeit bieten dagegen die mit Hilfe von POSIX Threads realisierten Implementierungen für gemeinsamen Adreßraum. Sie konnten auf dem gleichen System Speedup-Werte bis zu 23,0 erzielen. In einem Experiment konnten für Implementierung (E) für bis zu 8 Prozessoren sogar superlineare Speedups beobachtet werden. Vergleicht man die Laufzeiten der unterschiedlichen Implementierungsvarianten, unterscheiden sich die Laufzeiten der Implementierungen für gemeinsamen Adreßraum untereinander stärker als die Implementierungen für verteilten Adreßraum. Da sich die Implementierungen nur durch ihr Lokalitätsverhalten unterscheiden, deutet dies darauf hin, daß die Laufzeit der MPI-Implementierungen

[1] Andere Bezeichnung für Multibroadcastoperation, abgeleitet aus dem Namen der MPI-Operation `MPI_Allgather()`.

hauptsächlich durch die Laufzeit der MPI-Operationen bestimmt wird. Die POSIX-Threads-Implementierungen werden dagegen stark durch ihr Lokalitätsverhalten beeinflußt. Auf SMP-Clustersystemen kann die Skalierbarkeit durch die Verwendung hybrider Implementierungen verbessert werden, die MPI zur Kommunikation zwischen verschiedenen Knoten nutzen und auf jedem Knoten mehrere Threads ausführen, die über POSIX-Threads-Mechanismen synchronisiert werden. Da sich die Anzahl der erzeugten MPI-Prozesse und die Anzahl der erzeugten Threads so wählen lassen, daß die hybriden Implementierungen wie reine MPI- oder POSIX-Threads-Implementierungen arbeiten, liefern sie auf Parallelrechnern mit uniformer Speicherarchitektur nahezu die gleiche Performance wie die auf die entsprechende Speicherarchitektur spezialisierten Implementierungen.

5.4.6 Dynamische Lastbalancierung

In den bis dahin durchgeführten Laufzeitexperimenten konnten weder mit Hilfe von MPI realisierte Implementierungen für verteilten Adreßraum noch mit Hilfe von POSIX Threads realisierte Implementierungen für gemeinsamen Adreßraum eine zufriedenstellend hohe Skalierbarkeit erreichen. Es erscheint daher erforderlich, nach möglichen Ursachen zu suchen und Möglichkeiten zur Verbesserung der Skalierbarkeit zu entwickeln.

Da die Implementierungen für gemeinsamen Adreßraum in jeder Stufe mindestens eine Barrier-Operation zur Synchronisation nutzen, sonst jedoch nur implizit über Speicherzugriffe kommunizieren, kann eine mögliche Ursache für eine nichtoptimale Skalierbarkeit ein Lastungleichgewicht sein, das verhindert, daß alle Threads gleichzeitig die durch die Barrier-Operationen vorgegebenen Synchronisationspunkte erreichen. Ein solches Lastungleichgewicht kann beispielsweise durch eine ungleichmäßige Verteilung der Funktionsauswertungszeiten über die Komponenten des gewöhnlichen Differentialgleichungssystems, durch unterschiedliche Prozessorgeschwindigkeiten oder durch zur Laufzeit auftretende Ereignisse hervorgerufen werden. Um einem solchen Lastungleichgewicht entgegenzuwirken, können Verfahren zur dynamischen Lastbalancierung eingesetzt werden.

Eine Schwierigkeit besteht dabei jedoch darin, daß aufgrund der oft geringen Funktionsauswertungszeiten und der daraus resultierenden sehr feinen Granularität nur ein sehr geringer Overhead für die Lastbalancierung toleriert werden kann, damit der durch die Lastbalancierung erzielte Laufzeitgewinn den erforderlichen Overhead mehr als ausgleichen kann. Als Zielarchitektur eignen sich deshalb vor allem Parallelrechner mit gemeinsamem Adreßraum, da hier über den gemeinsamen Speicher ein effizienterer Datenaustausch möglich ist und spezielle atomare Maschineninstruktionen zur Synchronisation der Kontrollflüsse genutzt werden können. Für den Erfolg eines Lastbalancierungsverfahrens ist es dabei sehr wichtig, ein möglichst gutes Lokalitätsverhalten zu bewahren, da andernfalls die durch Speicherzugriffe entstehenden Wartezeiten den durch die Lastbalancierung erzielten Laufzeitgewinn zunichte machen würden.

Es wurden verschiedene Lastbalancierungsstrategien für Parallelrechner mit gemeinsamem Adreßraum realisiert und in Laufzeitexperimenten auf mehreren Rechnersystemen mit unterschiedlicher Architektur untersucht. Für Testprobleme, bei denen eine stark ungleichmäßige Verteilung der Funktionsauswertungskosten vorliegt, konnte dabei für alle untersuchten Lastbalancierungsstrategien eine Verbesserung der Skalierbarkeit erreicht werden. In bestimmten Fällen war es sogar möglich, für Probleme mit ausgewogener Verteilung der Funktionsauswertungszeiten einen signifikanten Laufzeitgewinn zu erzielen. Die eingesetzten Implementierungen wurden allerdings zunächst nur mit dem Ziel entwickelt, die Anwendbarkeit dynamischer Lastbalancierungsverfahren nachzuweisen. Es ist daher davon auszugehen, daß mit Hilfe alternativer Lastbalancierungsstrategien oder verbesserter Implementierungen noch höhere Laufzeitgewinne möglich sind.

5.5 Effiziente Implementierung für Gleichungen mit beschränkter Zugriffsdistanz

5.5.1 Definition und Bedeutung der beschränkten Zugriffsdistanz

Da im allgemeinen Fall von der Annahme ausgegangen werden muß, daß die Funktionsauswertung $f_j(t, \mathbf{w})$ auf alle Komponenten des Argumentvektors $\mathbf{w}$ zugreift, müssen allgemeine Implementierungen eingebet-

teter Runge-Kutta-Verfahren, die gewöhnliche Differentialgleichungssysteme mit beliebigen Abhängigkeiten unterstützen, alle s Stufen sequentiell berechnen. Dies führt dazu, daß das Lokalitätsverhalten durch die zeitliche Wiederverwendung von Vektoren mit der gleichen Dimension wie das zu integrierende Problem geprägt wird. Für Probleme mit großer Dimension hat dies die Konsequenz, daß nicht alle benötigten Vektoren im Cache gehalten werden können und dadurch die Anzahl der Cache-Fehlzugriffe ansteigt. Dies wiederum hat eine Verschlechterung der Laufzeit zur Folge. Lediglich durch ein Schleifen-Tiling können kleinere Arbeitsmengen erzeugt werden, die im Cache vorgehalten werden können und somit die Anzahl der Fehlzugriffe signifikant reduzieren können.

Bei einer Parallelisierung für verteilten Adreßraum muß darüber hinaus in jeder Stufe ein Austausch des jeweiligen Argumentvektors mit Hilfe einer Multibroadcastoperation durchgeführt werden. Werden bei einer Parallelisierung für gemeinsamen Adreßraum gemeinsame Datenstrukturen genutzt, müssen in jeder Stufe lediglich ein oder zwei Barrier-Operationen ausgeführt werden. Bei Verwendung verteilter Datenstrukturen ist jedoch ebenfalls ein Austausch des Argumentvektors mittels einer Multibroadcastoperation erforderlich. Dieser hohe Kommunikationsaufwand hat insbesondere auf Parallelrechnern mit verteiltem Adreßraum negative Auswirkungen auf die Skalierbarkeit.

Die Analyse des Lokalitätsverhaltens der allgemeinen Implementierungen zeigte, daß die Zugriffsdistanz einen wichtigen Einfluß auf die Speicherzugriffslokalität ausübt. Es liegt daher nahe, die Zugriffsdistanz des zu integrierenden Problems gezielt für Optimierungen auszunutzen. Dazu betrachten wir solche Probleme, deren Zugriffsdistanz klein gegenüber der Systemdimension ist, d. h. $d(\mathbf{f}) \ll n$. Wir sagen, diese Probleme besitzen eine *beschränkte Zugriffsdistanz*. Da für Probleme mit beschränkter Zugriffsdistanz die Funktionsauswertung $f_j(t, \mathbf{w})$ nur auf wenige Komponenten des Argumentvektors $\mathbf{w}$ zugreift, die sich alle innerhalb eines kleinen begrenzten Indexbereiches befinden, der auch j enthält, ergibt sich eine teilweise Entkoppelung der Stufen. Diese ermöglicht die Durchführung zusätzlicher Schleifentransformationen und kann zur Optimierung des Lokalitätsverhaltens sowie zur Reduzierung des Kommunikationsaufwandes paralleler Implementierungen ausgenutzt werden.

5.5.2 Blockbasierter Pipelining-Algorithmus

Das schlechte Lokalitätsverhalten der allgemeinen Implementierungen resultiert aus der sequentiellen Berechnungsreihenfolge der Stufen, die zu einer zeitlichen Wiederverwendung von Stufen- bzw. Argumentvektoren führte. Um den Arbeitsraum einer Iteration der äußersten Schleife zu verkleinern, müssen wir deshalb die äußere Schleife über die Systemdimension laufen lassen, so daß alle Schleifen, die über Stufen iterieren, innerhalb einer Iteration dieser Schleife ausgeführt werden. Die Entkoppelung der Stufen durch die Annahme einer beschränkten Zugriffsdistanz macht dies möglich. Da die Stufen nicht vollständig entkoppelt sind, muß die Berechnung der Stufen dabei jedoch ähnlich der Arbeitsweise einer Pipeline um mindestens $d(\mathbf{f})$ Komponenten versetzt erfolgen. Daraus ergibt sich ein Berechnungsschema, das durch die Ausführung mehrerer Pipelining-Schritte einen diagonalen Lauf über die Stufen- bzw. Argumentvektoren durchführt.

Diesem Vorgehen folgend, wurde eine blockweise arbeitende Implementierung auf Basis der Implementierung (D) realisiert und im Detail beschrieben. Diese Implementierung wird mit (PipeD) bezeichnet. Sie verwendet Blöcke der Größe $B \geq d(\mathbf{f})$. Die Berechnung der Blöcke erfolgt mit Hilfe von Präprozessormakros. Es kann jedoch eine Vielzahl weiterer Implementierungsvarianten realisiert werden. Beispielsweise können Unterprogrammaufrufe anstelle der Präprozessormakros verwendet werden, es können unterschiedliche Laufrichtungen realisiert werden, das Pipelining kann elementweise anstatt blockweise durchgeführt werden und es können alternative Basisimplementierungen verwendet werden. Bei großer Stufenzahl kann eine Aufsplittung in mehrere Pipelining-Läufe sinnvoll sein, um den Arbeitsraum der äußersten Schleife zu verkleinern. Wird eine konstante Schrittweite verwendet, kann das Pipelining über mehrere Zeitschritte ausgedehnt werden, was allerdings den Arbeitsraum der äußersten Schleife vergrößert.

5.5.3 Theoretische und experimentelle Analyse des Lokalitätsverhaltens

Das Lokalitätsverhaltens des Pipelining-Algorithmus wurde analog zur Untersuchung der allgemeinen Implementierungen mit Hilfe einer theoretischen Analyse sowie durch Simulations- und Laufzeitexperimente untersucht.

Die innersten Schleifen des blockbasierte Pipelining-Algorithmus dienen zur Verarbeitung einzelner Blöcke der Größe B. Ihre Schleifenstruktur entspricht weitestgehend derjenigen der Basisimplementierung (D), allerdings iterieren die Schleifen über die Systemdimension nur über B Komponenten. Dementsprechend ergeben sich die Größen der Arbeitsräume dieser Schleifen als Vielfache von B. Ausschlaggebend für das Lokalitätsverhalten sind jedoch die Arbeitsmengen der äußersten Schleife, deren Iterationen die Pipelining-Schritte realisieren. Wenn wir wie in der Basisimplementierung alle Argumentvektoren vollständig speichern, werden während eines Durchlaufs dieser Schleife alle Argumentvektoren sowie alle Hilfsvektoren benutzt. Daraus ergibt sich ein Arbeitsraum von $(s+3)n$, wie ihn auch Implementierung (D) besitzt. Kann dieser im Cache gespeichert werden, ist die zeitliche Wiederverwendung dieser Vektoren zwischen aufeinanderfolgenden Zeitschritten möglich. Eine Verbesserung des Lokalitätsverhaltens gegenüber der allgemeinen Basisimplementierung ergibt sich dadurch, daß der Arbeitsraum einer Iteration der äußersten Schleife des Berechnungskernels auf

$$\left(\frac{1}{2}s^2+\frac{7}{2}s+2\right)B=\Theta(s^2B)$$

reduziert wird. Da wir als kleinste Blockgröße $B=d(\mathbf{f})$ wählen können und $d(f)\ll n$ angenommen wird, ist die Größe dieses Arbeitsraumes geringer als der Speicherplatzbedarf für einen Vektor der Größe n. Alle allgemeinen Implementierungen sind dagegen auf die Wiederverwendung solcher Vektoren angewiesen, um eine vorhandene Cache-Hierarchie effizient ausnutzen zu können.

Simulationsexperimente, die mit Hilfe des Simulators Simics zur Untersuchung der tatsächlich auftretenden Arbeitsmengen durchgeführt wurden, bestätigen, daß Implementierung (PipeD) im Vergleich zu Implementierung (D) eine zusätzliche Arbeitsmenge besitzt, die durch die innerhalb eines Pipelining-Schrittes zugegriffenen Daten gebildet wird. Ist die Cachegröße hoch genug, um den Arbeitsraum eines Pipelining-Schrittes aufzunehmen, erzielt Implementierung (PipeD) eine deutlich geringere Zahl von Cache-Fehlzugriffen als Implementierung (D). Erst wenn die Cachegröße soweit gesteigert wird, daß mehrere Vektoren der Größe n im Cache gespeichert werden können, sinkt auch die Anzahl der Cache-Fehlzugriffe der Implementierung (D), so daß sich beide Implementierungen annähern. Auch für kleine Cachegrößen, die nicht ausreichend sind, um den Arbeitsraum eines Pipelining-Schrittes zu speichern, erzeugen Implementierung (PipeD) und deren Basisimplementierung (D) sehr ähnliche Cache-Fehlzugriffszahlen.

Diese Aussagen werden auch durch Messungen der bezüglich der Systemgröße und der Anzahl der Zeitschritte normierten Laufzeit auf einem Itanium-2-System bestätigt. Für die allgemeinen Implementierungen (D) und (Dblock) steigt ab einer bestimmten Systemgröße, ab der nicht mehr alle verwendeten Vektoren vollständig im Cache gespeichert werden können, die normierte Laufzeit stark an. Die normierte Laufzeit der Implementierung (PipeD) erhöht sich dagegen an dieser Stelle nur geringfügig. Sie kann deshalb für große Systemgrößen eine erheblich bessere normierte Laufzeit als (D) und (Dblock) erzielen.

5.5.4 Speicherplatzbedarf

Die vorgestellten Implementierungsvarianten des Pipelining-Algorithmus erfordern den gleichen Speicherplatzbedarf von $(s+3)n$ wie die Basisimplementierung (D), da sie die gleichen Datenstrukturen verwenden. Es ist jedoch möglich, den Pipelining-Algorithmus mit einem wesentlich geringeren Speicherplatzbedarf zu implementieren, da es nicht notwendig ist, alle Argumentvektoren und alle Hilfsvektoren vollständig zu speichern. Vielmehr genügt es, nur den Teil der Elemente der Argumentvektoren und der Hilfsvektoren im Speicher zu halten, der innerhalb eines Pipelining-Schrittes verwendet wird. Lediglich der Hilfsvektor $\Delta\eta$ muß vollständig gespeichert werden, da er als Ganzes zur Durchführung der Schrittweitenkontrolle benötigt wird. Unter Einbeziehung des Approximationsvektors η, der in jedem Fall vollständig gespeichert werden muß, ergibt sich ein Gesamtspeicherplatzbedarf von

$$2n+O\left(s^2B\right)\quad.$$

Der Speicherplatzbedarf ist vor allem bei der Integration sehr großer gewöhnlicher Differentialgleichungssysteme, wie sie z. B. durch die Ortsdiskretisierung von partiellen Differentialgleichungssystemen entstehen, von großer Bedeutung. Hier kann der Speicherplatzbedarf für mehrstufige allgemeine Implementierungen inakzeptabel hoch sein. Es wurden deshalb von verschiedenen Autoren spezielle explizite

Runge-Kutta-Verfahren vorgeschlagen, die einen Speicherplatzbedarf von $2n$ besitzen. Die vorgeschlagenen Verfahren erreichen diese Reduzierung des Speicherplatzbedarfs durch die Verwendung einer nicht vollbesetzten Verfahrensmatrix, die besonderen Anforderungen genügen muß. Aufgrund der sich daraus ergebenden Einschränkungen der numerischen Eigenschaften sind diese Verfahren nur für bestimmte Klassen von Anfangswertproblemen geeignet.

Unter der Annahme, daß sowohl die Blockgröße B als auch die Stufenzahl s klein gegenüber der Systemgröße n ist, benötigt der in dieser Arbeit vorgeschlagene Pipelining-Algorithmus nur unwesentlich mehr Speicherplatz als die von anderen Autoren vorgeschlagenen $2n$-Verfahren. Er ermöglicht im Gegensatz zu diesen jedoch eine effiziente Schrittweitenkontrolle durch die Verwendung eingebetteter Lösungen. Würde man auf eine Schrittweitenkontrolle verzichten, könnte der Speicherplatzbedarf sogar auf $n + O\left(s^2 B\right)$ verringert werden. Weiterhin erfordert der Pipelining-Algorithmus keine Einschränkungen bezüglich der Belegung der Verfahrensmatrix. Ein Nachteil besteht jedoch darin, daß er nur auf Probleme mit beschränkter Zugriffsdistanz anwendbar ist.

5.5.5 Frühzeitiger Abbruch zu verwerfender Zeitschritte

Zeitschritte, die verworfen werden müssen, können während der Ausführung des Pipelining-Algorithmus frühzeitig erkannt werden, bevor der diagonale Lauf über die Stufen abgeschlossen ist. Dies wird dadurch ermöglicht, daß der Pipelining-Algorithmus in jedem Pipelining-Schritt eine oder mehrere Komponenten des Fehlervektors berechnet. Sobald bekannt ist, daß der lokale Fehler eine vorgegebene Toleranz überschreitet, steht fest, daß der aktuell berechnete Zeitschritt verworfen werden muß und zu einem geeigneten Zeitpunkt abgebrochen werden kann. Dadurch kann Rechenzeit für diesen Zeitschritt eingespart werden, man muß jedoch in Kauf nehmen, daß unter Umständen der lokale Fehler zu klein geschätzt wird, da nicht alle Fehlerkomponenten bekannt sind. Man kann dem durch Anwendung einer Heuristik entgegenwirken, indem beispielsweise in jedem Zeitschritt mindestens bis zu dem Index gerechnet wird, an dem im vorhergehenden Zeitschritt die Komponente des lokalen Fehlers den höchsten Wert hatte. In Laufzeitexperimenten, die zur Untersuchung einer auf diese Weise modifizierten Schrittweitenkontrolle durchgeführt wurden, konnte jedoch kein oder nur ein geringer Laufzeitgewinn erzielt werden. Dies war dadurch bedingt, daß die verwendete Schrittweitenkontrolle nur eine kleine Zahl verworfener Zeitschritte erzeugte und dadurch das Einsparungspotential gering war, während sich gleichzeitig der durch das Prüfen der Abbruchbedingung entstehende Overhead negativ auf die Laufzeit auswirkte.

5.5.6 Parallele Implementierung

Auch bei der Erstellung paralleler Implementierungen kann eine beschränkte Zugriffsdistanz zur Verbesserung des Lokalitätsverhaltens ausgenutzt werden. Darüber hinaus ermöglicht sie aber auch eine effizientere Realisierung der Kommunikation zwischen den beteiligten Kontrollflüssen.

Allgemeine Implementierungen für verteilten Adreßraum mußten in jeder Stufe eine Multibroadcastoperation ausführen, um den jeweiligen Argumentvektor allen beteiligten Prozessen zur Verfügung zu stellen. Besitzt das zu integrierende Problem jedoch eine beschränkte Zugriffsdistanz, ist während der Stufenberechnung Kommunikation nur mit benachbarten Prozessen notwendig, und es können Einzeltransferoperationen genutzt werden, um diesen Datenaustausch zu realisieren. Dabei ist es zumindest teilweise möglich, die Datenübertragungszeiten der Einzeltransferoperationen durch Berechnungen zu überdecken. In dieser Arbeit wurden verschiedene Implementierungsvarianten vorgestellt, die mit Hilfe von nichtblockierenden Einzeltransferoperationen kommunizieren und die Stufen entweder nacheinander berechnen oder das Pipelining-Berechnungsschema zusammen mit verschiedenen Finalisierungsvarianten zur Verbesserung des Lokalitätsverhaltens nutzen.

Bei der Implementierung für gemeinsamen Adreßraum bietet eine beschränkte Zugriffsdistanz ebenfalls die Möglichkeit, den Kommunikations- bzw. Synchronisationsaufwand zu reduzieren. Die Datenstrukturen können dabei entweder als verteilte oder gemeinsame Datenstrukturen realisiert werden. Werden verteilte Datenstrukturen genutzt, erfolgt der Datenaustausch auf ähnliche Weise wie im Fall der Implementierungen für verteilten Adreßraum, wobei die dort verwendeten Kommunikationsoperationen durch das Kopieren von Speicherbereichen innerhalb des gemeinsamen Adreßraumes emuliert werden. Die in dieser Arbeit betrachteten Implementierungsvarianten nutzen die verwendeten Vektoren als gemeinsame

Datenstrukturen, auf die alle beteiligten Threads zugreifen können. Ein expliziter Datenaustausch ist daher nicht erforderlich, es muß jedoch eine Synchronisation der Threads mit Hilfe von Barrier-Operationen oder Mutex-Variablen durchgeführt werden. Die Berechnungsreihenfolge, wie sie zur Realisierung der Implementierungen für verteilten Adreßraum verwendet wurde, kann aber beibehalten werden.

5.5.7 Laufzeitverhalten auf verschiedenen Rechnersystemen

Zur Untersuchung der sequentiellen Implementierungsvarianten des Pipelining-Algorithmus wurde eine Reihe von Einzelexperimenten mit den Testproblemen MEDAKZO und BRUSS2D unter Verwendung unterschiedlicher Systemgrößen auf verschiedenen Zielplattformen durchgeführt. In diesen Experimenten konnte durch Verwendung des Pipelining-Ansatzes in der Regel eine Laufzeitverbesserung erzielt werden, wenn die Systemgröße hoch genug war, so daß nicht alle verwendeten Vektoren im Cache gespeichert werden konnten.

Die parallelen Implementierungsvarianten wurden hinsichtlich ihrer Skalierbarkeit auf verschiedenen Zielplattformen mit unterschiedlicher Architektur untersucht. Dabei zeigte sich, daß die spezialisierten Implementierungen für verteilten Adreßraum in der Lage sind, erheblich höhere Speedups als die allgemeinen Implementierungen für verteilten Adreßraum zu erzielen. Einen wesentlichen Einfluß auf die Skalierbarkeit hat dabei der erforderliche Kommunikationsaufwand. Insbesondere auf Parallelrechnern mit gemeinsamem Adreßraum konnte jedoch auch ein signifikanter Einfluß des Lokalitätsverhaltens beobachtet werden.

Auch die spezialisierten Implementierungen für gemeinsamen Adreßraum können höhere Speedups als die allgemeinen Implementierungen erzielen. Die Laufzeitunterschiede zwischen den spezialisierten und den allgemeinen Implementierungen sind allerdings weniger groß als im Fall der Implementierungen für verteilten Adreßraum. Im Vergleich zu den spezialisierten Implementierungen für verteilten Adreßraum waren die spezialisierten Implementierungen für gemeinsamen Adreßraum teilweise sogar langsamer. Dies deutet darauf hin, daß die Implementierungen für gemeinsamen Adreßraum durch die gemeinsame Nutzung der Vektoren durch alle beteiligten Threads die NUMA-Architektur der verwendeten Zielsysteme nicht optimal ausnutzen. Daß das Lokalitätsverhalten gerade auf Parallelrechnern mit gemeinsamem Adreßraum einen wichtigen Einfluß auf die Skalierbarkeit ausübt, kann man insbesondere am Verhalten der spezialisierten Implementierung (Dbc) ablesen, deren Lokalitätsverhalten nahezu identisch mit dem der allgemeinen Implementierung (D) ist, die bezüglich der verwendeten Synchronisationsmechanismen jedoch weitestgehend der Pipelining-Implementierung (PipeD) entspricht. In vielen Experimenten stimmt die Laufzeit der Implementierung (Dbc) mit der Laufzeit der Implementierung (D) überein, während die Pipelining-Implementierung (PipeD) eine deutlich bessere Laufzeit erzielen kann. Bei Verwendung verteilter Datenstrukturen in Kombination mit einem expliziten, durch das Kopieren von Speicherbereichen realisierten Datenaustausch sind daher mindestens ebenso gute Speedups wie für die MPI-Implementierungen zu erwarten. In einigen Experimenten sind jedoch auch die verwendeten Synchronisationsmechanismen ausschlaggebend. Dies ist insbesondere bei der Integration von Problemen mit relativ geringer Systemgröße bei Verwendung einer hohen Prozessoranzahl der Fall. In diesen Experimenten arbeitet (Dbc) deutlich schneller als (D) und erreicht oder übertrifft eventuell sogar Implementierung (PipeD).

Hybride Implementierungen, die zur Ausnutzung der spezifischen Architektur von SMP-Clustersystemen verschiedene Programmiermodelle, z. B. MPI und POSIX Threads, kombinieren, wurden im Rahmen einer Bachelor-Arbeit (Behrend 2005) realisiert und untersucht. Diese Implementierungen konnten jedoch keine wesentliche Laufzeitverbesserung gegenüber reinen MPI-Implementierungen erzielen. Als eine der möglichen Ursachen wurde ein Lastungleichgewicht identifiziert, das dadurch hervorgerufen wird, daß Threads, die MPI-Operationen ausführen, eine höhere Laufzeit benötigen als andere Threads.

5.5.8 Dynamische Lastbalancierung

In Experimenten mit allgemeinen Implementierungsvarianten wurde bereits gezeigt, daß der Einsatz von Lastbalancierungsverfahren innerhalb von Implementierungen eingebetteter Runge-Kutta-Verfahren lohnenswert sein kann. Trotz des zusätzlichen Overheads, der für die Überwachung der Lastverteilung und für die Migration von Tasks erforderlich ist, macht sich die Verwendung von Lastbalancierungstechniken insbesondere dann bezahlt, wenn eine stark ungleichförmige Verteilung der Funktionsauswertungskosten

vorliegt. Aber auch für Probleme mit nahezu ausgeglichener Verteilung der Funktionsauswertungskosten konnten in mehreren Experimenten zumindest geringe Laufzeitverbesserungen erzielt werden.

Auch für auf eine beschränkte Zugriffsdistanz spezialisierte Implementierungen erscheint der Einsatz von Lastbalancierungsverfahren sehr vielversprechend. Insbesondere für hybride Implementierungsvarianten ist durch den Einsatz von Lastbalancierungstechniken eine signifikante Verbesserung der Laufzeit zu erwarten, da ohne Lastbalancierung Threads, die keine MPI-Operationen ausführen, auf Threads, die mittels MPI mit anderen Knoten kommunizieren, warten müssen. Im Unterschied zu den allgemeinen Implementierungen ist es bei den spezialisierten Implementierungen nicht erforderlich, die Stufen nacheinander zu berechnen und die Berechnung der einzelnen Stufen durch Barrier-Operationen zu trennen. Stattdessen kann der gesamte Berechnungskernel durch eine einzige Lastbalancierungsphase realisiert werden. Voraussetzung dafür ist allerdings, daß die Datenabhängigkeiten der Funktionsauswertungen durch entsprechende Abhängigkeiten zwischen Tasks modelliert werden.

Durch die Auswahl der Scheduling-Strategie kann das Lokalitätsverhalten beeinflußt werden. Führt die Scheduling-Strategie eine Breitensuche auf dem Taskgraphen durch, ergibt sich ein Lokalitätsverhalten ähnlich dem der stufenweise arbeitenden allgemeinen Implementierungen. Wird dagegen eine Tiefensuche auf dem Taskgraphen durchgeführt resultiert ein Lokalitätsverhalten ähnlich dem des Pipelining-Algorithmus.

5.5.9 Ansatz zur automatischen Spezialisierung

Die im Rahmen dieser Arbeit durchgeführten Experimente belegen, daß durch die Spezialisierung bezüglich des Problems eine Verbesserung des Lokalitätsverhaltens und der Skalierbarkeit eingebetteter Runge-Kutta-Verfahren erreicht werden kann. Andere Autoren verfolgen die Strategie einer Spezialisierung bezüglich des Verfahrens, indem sie spezielle Verfahrenskoeffizienten vorschlagen bzw. verwenden, die bestimmte Vorteile bei einer Implementierung bieten, wie z. B. eine Entkoppelung der Stufen, die zur Parallelisierung ausgenutzt werden kann. Die Spezialisierung – sowohl bezüglich des Problems als auch bezüglich des Verfahrens – ist also im allgemeinen ein erfolgreiches Konzept, das eine effiziente Nutzung moderner Parallelrechner erst möglich macht.

Statische Scheduling-Verfahren könnten ein möglicher Weg sein, um eine Spezialisierung bezüglich eines gegebenen Problems und eines gegebenen Verfahren automatisch durchzuführen. Die Idee dieses Ansatzes besteht darin, zunächst einen Graphen zu konstruieren, der die durch die Funktionsauswertung und die Verfahrenskoeffizienten gegebenen Abhängigkeiten beschreibt. Dieser Graph kann als Eingabe für ein Scheduling-Verfahren dienen, das versucht, einen parallelen Ablaufplan mit geringer Latenz zu erzeugen, die durch ein gutes Lokalitätsverhalten und geringe Kommunikationskosten erreicht wird.

5.5.10 Übertragbarkeit auf andere Lösungsverfahren

Die Ausnutzung einer beschränkten Zugriffsdistanz ist nicht nur für eingebettete Runge-Kutta-Verfahren möglich. Als Beispiel wurden im Rahmen einer Bachelor-Arbeit (Gerkhardt 2005) Möglichkeiten zur Lokalitätsoptimierung sequentieller iterierter Runge-Kutta-Verfahren untersucht. Die dabei durchgeführten Laufzeitexperimente bestätigten, daß durch diesen Ansatz auch für iterierte Runge-Kutta-Verfahren eine Verbesserung des Lokalitätsverhalten erreicht werden kann. Es ist davon auszugehen, daß auch für viele weitere Verfahren die Ausnutzung einer beschränkten Zugriffsdistanz oder allgemein eine Spezialisierung bezüglich des Problems zur Verbesserung des Lokalitätsverhaltens und der Skalierbarkeit ausgenutzt werden kann.

5.6 Ausblick

In dieser Arbeit wurden verschiedene Ansätze zur effizienten Implementierung eingebetteter Runge-Kutta-Verfahren vorgestellt, diskutiert und untersucht. Aufgrund der Komplexität dieses Themengebietes verbleiben jedoch eine Reihe von Fragestellungen, die einer ausführlichen Untersuchung im Rahmen zukünftiger Forschungsarbeiten bedürfen. Darüber hinaus sind durch diese Arbeit verschiedene neue Aufgabenstellungen entstanden, die über das Thema der Arbeit hinausgehen.

- **Ausrichtung der Datenstrukturen bezüglich der Abbildung auf Cachezeilen:** Es wurde aufgezeigt, daß die Ausrichtung der Datenstrukturen bezüglich der Abbildung auf Cachezeilen einen wichtigen Einfluß auf das Lokalitätsverhalten ausübt. Insbesondere wurde an einem Beispiel verdeutlicht, daß eine ungünstige Anordnung zu Thrashing und somit zu einer Steigerung der Anzahl der Cache-Fehlzugriffe führen kann. Weiterhin wurden Richtlinien angegeben, die helfen, eine solche ungünstige Anordnung zu vermeiden. Im Rahmen zukünftiger Forschung sollten die Auswirkungen der Ausrichtung der Datenstrukturen im Detail untersucht werden. Ziel sollte es sein, eine Vorschrift für die Ausrichtung der Datenstrukturen angeben zu können, die eine Minimierung der Anzahl der Cache-Fehlzugriffe ermöglicht.

- **Automatische Auswahl von Implementierungsvarianten:** In dieser Arbeit wurden verschiedene Implementierungsvarianten eingebetteter Runge-Kutta-Verfahren vorgestellt, die ein unterschiedliches Lokalitätsverhalten besitzen oder die Kommunikation zwischen Kontrollflüssen auf unterschiedliche Weise realisieren. Die durchgeführten Experimente zeigen, daß sich die Laufzeiten der betrachteten Implementierungsvarianten zum Teil erheblich unterscheiden. Die Auswahl einer geeigneten Implementierungsvariante ist daher wichtig, um moderne Rechnersysteme effizient nutzen zu können. Der Aufwand für eine manuelle Auswahl einer Implementierungsvariante ist allerdings sehr hoch, da viele Faktoren, u. a. die Architektur der Zielplattform, die Systemgröße und das Zugriffsverhalten des zu integrierenden Problems sowie die Stufenzahl des verwendeten Verfahrens, einen Einfluß auf die Laufzeit der Implementierungsvarianten ausüben. Es sollten deshalb Möglichkeiten zur automatischen Auswahl einer effizienten Implementierungsvariante untersucht werden.

  Dazu sollte zunächst festgestellt werden, ob eine beschränkte Zugriffsdistanz gegeben ist, damit bekannt ist, ob spezialisierte Implementierungsvarianten eingesetzt werden können oder ob ausschließlich allgemeine Implementierungsvarianten nutzbar sind. Danach könnte mit Hilfe von statisch oder dynamisch gewonnenen Informationen eine automatische Auswahl der günstigsten Implementierungsvariante durchgeführt werden.

  1. **Statischer Ansatz:** Wichtige Informationen über das zu integrierende Problem und das zu verwendenden Verfahren werden vom Nutzer erfragt bzw. anhand einer Datenbasis ermittelt. Zur Entscheidungsfindung notwendige Parameter der Speicherhierarchie werden vom Betriebssystem erfragt, von der Hardware des Zielsystems ausgelesen oder durch Messungen bestimmt. Anhand dieser Informationen wählt ein Expertensystem eine geeignete Implementierung aus.
  2. **Dynamischer Ansatz:** Während des Integrationsvorgangs werden zur Berechnung der ersten Zeitschritte verschiedene Implementierungsvarianten genutzt und deren Laufzeit bestimmt. Die schnellste Implementierungsvariante wird für die Berechnung der verbleibenden Zeitschritte verwendet.

  Zwar kann voraussichtlich nicht garantiert werden, das ein solcher Automatismus unter allen Umständen die beste Implementierungsvariante auswählt, ein Nutzer, der häufig verschiedene Integrationen auf unterschiedlichen Zielsystemen durchführt, wird jedoch insgesamt eine Zeitersparnis gegenüber dem Fall erreichen, daß er immer die gleiche Implementierungsvariante nutzt.

- **Implementierungen für gemeinsamen Adreßraum mit verteilten Datenstrukturen:** Alle in dieser Arbeit untersuchten Implementierungsvarianten für gemeinsamen Adreßraum nutzen gemeinsame Datenstrukturen, die von einem Thread allokiert und von allen Threads benutzt werden. Die Tatsache, daß in einigen Experimenten auf Rechnersystemen mit NUMA-Architektur Implementierungsvarianten, die für einen verteilten Adreßraum entworfen wurden, eine geringere Laufzeit erreichten, deutet jedoch darauf hin, daß die Verwendung gemeinsamer Datenstrukturen nicht zu einer optimalen Ausnutzung der Speicherhierarchie führt. Es erscheint daher lohnenswert, Implementierungsvarianten zu realisieren und zu untersuchen, die verteilte Datenstrukturen verwenden, zwischen denen ein expliziter Datenaustausch erfolgt. Aufgrund der Ergebnisse der bereits durchgeführten Untersuchungen ist zu erwarten, daß solche Implementierungen trotz des durch den expliziten Datenaustausch entstehenden Overheads aufgrund der verbesserten Lokalität innerhalb des Hauptspeichers eine geringere Laufzeit erreichen können. Interessant wäre eine Untersuchung derartiger Implementierungsvarianten insbesondere auf NUMA-Systemen mit großer Prozessoranzahl, wie z. B. der SGI Altix 4700, die am Leibniz Rechenzentrum München betrieben wird und seit Juni 2006 für externe Projekte zur Verfügung steht.

- **Hybride Implementierungen für SMP-Cluster:** Implementierungen, die verschiedene Programmiermodelle kombinieren, um die Speicherarchitektur von SMP-Clustern auszunutzen, wurden im Rahmen einer Bachelor-Arbeit (Behrend 2005) untersucht. In den dabei erstellten Implementierungen werden gemeinsame Datenstrukturen durch mehrere Threads genutzt. Sie arbeiten deshalb auf Rechnersystemen mit stark ungleichförmigen Speicherzugriffszeiten nicht optimal. Hinzu kommt, daß diejenigen Threads, die MPI-Operationen ausführen, eine längere Laufzeit benötigen als die übrigen Threads, so daß die übrigen Threads auf diese warten müssen. Diese Implementierungen können voraussichtlich verbessert werden, indem verteilte Datenstrukturen genutzt werden und durch eine dynamische Arbeitsverteilung ein Ausgleich zwischen den Threads, die MPI-Operationen ausführen, und den übrigen Threads herbeigeführt wird.

- **Implementierung des Pipelining-Algorithmus mit reduziertem Speicherplatzbedarf:** Der Pipelining-Algorithmus kann mit Speicherplatz $2n + O\left(s^2 B\right)$ realisiert werden. Neben der Speicherplatzersparnis hat eine solche Implementierung den Vorteil einer höheren Lokalität. Bisher wurde keine Implementierung des Pipelining-Algorithmus mit reduziertem Speicherplatzbedarf realisiert und untersucht. Alle in dieser Arbeit betrachteten Implementierungen des Pipelining-Algorithmus benutzen die gleichen Datenstrukturen wie die allgemeine Basisimplementierung, von der sie abgeleitet wurden, und benötigen daher einen Speicherplatz von $O(sn)$. In der Zukunft soll deshalb der Pipelining-Algorithmus mit reduziertem Speicherplatzbedarf implementiert und untersucht werden. Interessant ist in diesem Zusammenhang nicht nur eine interne Implementierung, die vollständig im Hauptspeicher arbeitet, sondern auch eine externe Implementierung, welche die Vektoren $\eta_{\kappa+1}$ und $\hat{\eta}_{\kappa+1}$ auf ein externes Speichermedium auslagert und somit nur einen Hauptspeicherbedarf von $O\left(s^2 B\right)$ besitzt.

- **Realisierung des Pipelining-Algorithmus in Kombination mit Richardson-Extrapolation**: Bei Durchführung der Schrittweitenkontrolle mittels Richardson-Extrapolation werden in jedem Zeitschritt drei Integrationsschritte ausgeführt: Ein Schritt $t_\kappa \rightarrow t_{\kappa+1}$ mit Schrittweite h_κ zur Bestimmung der Approximation $\eta_{\kappa+1}^{(h_\kappa)}$ sowie zwei aufeinanderfolgende Schritte $t_\kappa \rightarrow t_\kappa + \frac{h_\kappa}{2} \rightarrow t_{\kappa+1}$ mit Schrittweite $\frac{h_\kappa}{2}$ zur Bestimmung der Approximation $\eta_{\kappa+1}^{(2\times h_\kappa/2)}$. Dabei ist es generell möglich, $\eta_{\kappa+1}^{(h_\kappa)}$ und $\eta_{\kappa+1}^{(2\times h_\kappa/2)}$ parallel zu bestimmen. Der Pipelining-Algorithmus erlaubt es zusätzlich, die beiden Schritte $t_\kappa \rightarrow t_\kappa + \frac{h_\kappa}{2}$ und $t_\kappa + \frac{h_\kappa}{2} \rightarrow t_{\kappa+1}$ parallel auszuführen. Durch Einsatz zusätzlicher Prozessoren ist es somit möglich, eine Schrittweitenkontrolle auf Basis der Richardson-Extrapolation in etwa der gleichen Zeit durchzuführen, wie eine Schrittweitenkontrolle auf Basis eingebetteter Lösungen. Der Vorteil dieses Vorgehens gegenüber der Verwendung eingebetteter Lösungen besteht darin, daß eine größere Freiheit bezüglich der Wahl der Verfahrenskoeffizienten gegeben ist. Dies kann in bestimmten Fällen eine Reduzierung der Anzahl der Zeitschritte oder eine genauere Bestimmung der Lösungsfunktion ermöglichen.

- **Extrapolationsverfahren:** Führt man den eben beschriebenen Gedanken fort, indem weitere Approximationen $\eta_{\kappa+1}^{(3\times h_\kappa/3)}, \eta_{\kappa+1}^{(4\times h_\kappa/4)}, \eta_{\kappa+1}^{(5\times h_\kappa/5)}, \ldots$ parallel berechnet werden, erhält man ein Extrapolationsverfahren mit hohem Parallelitätsgrad, das zusätzlich eine einfache Möglichkeit zur Ordnungssteuerung bietet.

- **Frühzeitiger Abbruch zu verwerfender Zeitschritte:** Der Pipelining-Algorithmus erlaubt es, Zeitschritte, die verworfen werden müssen, frühzeitig zu erkennen. Man kann daher versuchen, durch deren Abbruch Rechenzeit einzusparen. In durchgeführten Experimenten konnte jedoch kaum ein Laufzeitgewinn erzielt werden, da nur wenige Zeitschritte verworfen wurden und folglich das Einsparungspotential geringer war als der zum Prüfen der Abbruchbedingung erforderliche Overhead. In einigen Experimenten konnte jedoch ein geringer Laufzeitgewinn erzielt werden, was zeigt, daß dieses Vorgehen ein prinzipiell sinnvoller Ansatz zur Laufzeitverbesserung ist. Es sollte deshalb untersucht werden, ob für alternative Strategien der Schrittweitenkontrolle, die eine größere Zahl von Zeitschritten verwerfen, ein zuverlässiger Laufzeitgewinn erzielt werden kann. Darüber hinaus sollten alternative Implementierungsvarianten betrachtet werden, die im Fall des Verwerfens eines Zeitschrittes einen höheren Laufzeitgewinn erzielen, indem sie die Pipeline gezielt in dem Indexbereich aufbauen, in dem der höchste lokale Fehler vermutet wird.

- **Dynamische Lastbalancierungsverfahren:** Für allgemeine Implementierungen eingebetteter Runge-Kutta-Verfahren wurde untersucht, ob der Einsatz dynamischer Lastbalancierungsverfahren zu einem Laufzeitgewinn führen kann. Die durchgeführten Laufzeitexperimente zeigen, daß unter bestimmten Voraussetzungen sogar für Probleme mit nahezu gleichförmig verteilten Funktionsauswertungszeiten ein Laufzeitgewinn erzielt werden kann. Eine weitere Untersuchung der Einsatzmöglichkeiten dynamischer Lastbalancierungsverfahren erscheint daher insbesondere in folgenden Bereichen lohnenswert:

  1. Verbesserung der für den allgemeinen Fall realisierten Lastbalancierungsstrategien,
  2. Entwurf und Implementierung von Lastbalancierungstechniken für Probleme mit beschränkter Zugriffsdistanz,
  3. Entwurf und Implementierung von Lastbalancierungsstrategien für hybride Implementierungen, die MPI und POSIX Threads kombinieren und
  4. Untersuchung der Anwendbarkeit von Lastbalancierungsverfahren auf andere numerische Anwendungen.

- **Statische Scheduling-Verfahren:** In dieser Arbeit wurde vorgeschlagen, statische Scheduling-Verfahren einzusetzen, um eine automatische Spezialisierung sowohl bezüglich des Problems als auch bezüglich des Verfahrens durchzuführen. Zur Motivation wurde dazu das prinzipielle Vorgehen zur Umsetzung dieses Ansatzes skizziert. Eine Aufgabe für zukünftige Forschungsarbeiten ist die Entwicklung eines Scheduling-Verfahrens, das zur Parallelisierung und zur Optimierung des Lokalitätsverhaltens eine Ablaufplanung anhand eines erweiterten Datenflußgraphen vornimmt.

- **Übertragung auf andere Lösungsverfahren:** Im Rahmen einer Bachelor-Arbeit (Gerkhardt 2005) wurde gezeigt, daß eine beschränkte Zugriffsdistanz auch bei der Implementierung sequentieller iterierter Runge-Kutta-Verfahren zur Optimierung des Lokalitätsverhaltens ausgenutzt werden kann. Aufgrund des Zeitrahmens konnten jedoch nicht alle Optimierungsmöglichkeiten ausgeschöpft werden. So sind noch mehrere Variationen der Schleifenstruktur iterierter Runge-Kutta-Verfahren möglich, die das Lokalitätsverhalten weiter verbessern können. Weiterhin kann eine beschränkte Zugriffsdistanz auch bei der Parallelisierung iterierter Runge-Kutta-Verfahren ausgenutzt werden. Solche Implementierungen sollen im Rahmen zukünftiger Arbeiten realisiert und untersucht werden. Darüber hinaus soll untersucht werden, für welche weiteren Lösungsverfahren die Ausnutzung einer beschränkten Zugriffsdistanz möglich ist.

A Verwendete Rechnersysteme

Im folgenden werden verschiedene sequentielle und parallele Rechnersysteme beschrieben, die im Rahmen dieser Arbeit für die Durchführung von Laufzeitexperimenten genutzt wurden.

A.1 Fujitsu-Siemens SCENIC W600

Zur Untersuchung der sequentiellen Implementierungsvarianten wurde unter anderem eine Workstation des Typs Fujitsu-Siemens SCENIC W600 eingesetzt, die mit einem Pentium-4-Prozessor (Northwood-Kern) mit einer Taktfrequenz von 3 GHz und einem Hauptspeicher von 1 GB ausgestattet ist. In Tab. A.1 sind die Cache-Parameter des Prozessors zusammengefaßt.

Stufe	Typ	Größe	Assoziativität	Zeilengröße	Latenz
1	Daten	8 KB	4-fach	64 Byte	2 Takte
1	Instruktionen	12 K µ-Operationen	8-fach	—	—
2	Unified	512 KB	8-fach	64 Byte	7 Takte

Tab. A.1: Cache-Parameter des Intel Pentium 4 (Northwood). Quelle: Intel (2006a).

A.2 Sun Blade 1000

Weitere Untersuchungen der sequentiellen Implementierungsvarianten wurden auf einer SMP-Workstation des Typs Sun Blade 1000 durchgeführt. Die verwendete Maschine ist mit zwei Prozessoren des Typs UltraSPARC III Cu ausgestattet, die mit einer Taktfrequenz von 1050 MHz arbeiten. Die Prozessoren besitzen eine zweistufige Cache-Hierarchie, deren Parameter in Tabelle A.2 dargestellt sind.

Stufe	Typ	Größe	Assoziativität	Zeilengröße	Latenz	Rückschreibstrategie
1	Daten	64 KB	4-fach	32 Byte	2 Takte	Write Through
1	Instruktionen	32 KB	4-fach	32 Byte	2 Takte	Write Invalidate
2	Unified	8 MB	2-fach	512 Byte	≈19 Takte	Write Back

Tab. A.2: Cache-Parameter des Sun UltraSPARC III Cu. Quelle: Sun (2003a).

A.3 HP Integrity rx5670

Der HP Integrity rx5670 ist ein SMP-Server auf Basis des Itanium-2-Prozessors. Das für die Durchführung von Laufzeitexperimenten verwendete System ist mit der maximal einsetzbaren Anzahl von 4 Prozessoren bestückt und verfügt über insgesamt 8 GB Hauptspeicher. Die Prozessoren arbeiten mit einer Taktfrequenz von 1,5 GHz. Sie besitzen eine für die Durchführung von Untersuchungen des Lokalitätsverhaltens interessante dreistufige Cache-Hierarchie, deren Parameter in Tab. A.3 zusammengefaßt sind. Der Datenaustausch zwischen den Prozessoren und dem Chipsatz, der die Schnittstelle zu den Speichermodulen und zu I/O-Geräten bildet, erfolgt über ein Bussystem (*Front Side Bus*) mit einer Übertragungsbandbreite von 6,4 GB/s.

Stufe	Typ	Größe	Assoziativität	Zeilengröße	Latenz
1	Daten	16 KB	4-fach	64 Byte	1 Takt
1	Instruktionen	16 KB	4-fach	64 Byte	1 Takt
2	Unified	256 KB	8-fach	128 Byte	5 Takte
3	Unified	6 MB	12-fach	128 Byte	14 Takte

Tab. A.3: Cache-Parameter des Intel Itanium 2 mit 6 MB L3-Cache. Quelle: Intel (2004).

A.4 Sun Fire 6800

Die Sun Fire 6800 ist ein SMP-Server, der mit bis zu 24 UltraSPARC-III-Prozessoren ausgestattet werden kann. Das für die Durchführung von Laufzeitexperimenten verwendete System wird am Universitätsrechenzentrum der Martin-Luther-Universität Halle-Wittenberg betrieben. Es ist mit 24 Prozessoren des Typs UltraSPARC III Cu mit einer Taktfrequenz von 900 MHz ausgestattet und verfügt über 24 GB Hauptspeicher. Die Cache-Hierarchie ist zweistufig. Tabelle A.2 zeigt deren wichtigste Parameter. Die Prozessoren verfügen über einen integrierten Speichercontroller, so daß Speichermodule direkt an den Prozessor angebunden werden können. Die Verbindung zwischen verschiedenen Prozessoren erfolgt über ein hierarchisch aufgebautes Verbindungsnetzwerk, den sogenannten *Fireplane Interconnect* (Charlesworth 2001; Sun 2003b). Es verbindet je zwei CPUs und deren Speichermodule über einen *Dual CPU Data Switch* (DCDS). Zwei solcher CPU-Paare befinden sich auf einem Mainboard. Auf jedem Mainboard befindet sich auch ein Datenswitch, der die zwei Dual CPU Data Switches dieser beiden CPU-Paare verbindet. Das Gesamtsystem ist aus insgesamt 6 solcher Mainboards aufgebaut, die wiederum über Datenswitches, die sich auf sogenannten *Fireplane Switch Boards* befinden, verbunden sind. Durch den hierarchischen Aufbau des Verbindungsnetzwerks ergeben sich unterschiedliche Speicherzugriffszeiten, die davon abhängen, wieviele Hierarchieebenen eine Anfrage eines Prozessors bis zum Erreichen des betreffenden Speichermoduls durchlaufen muß. Tabelle A.4 gibt einen Überblick.

Zugriffsort	Pin-zu-Pin-Latenz
Lokaler Speicher	180 ns
Speicher auf gleichem Board über DCDS	193 ns
Speicher auf anderer Seite des gleichen Boards	207 ns
Speicher auf anderem Board	240 ns

Tab. A.4: Speicherzugriffslatenzen einer Sun Fire 6800. Quelle: Sun (2003b).

A.5 Clustersysteme aus Standardkomponenten

Aufgrund ihres günstigen Preis-/Leistungsverhältnisses haben Clustersysteme aus Standardkomponenten eine weite Verbreitung gefunden. Die in dieser Arbeit präsentierten Experimente wurden u. a. auf zwei solchen Systemen durchgeführt.

Das ältere der beiden Systeme ist der Chemnitzer Linux Cluster (CLiC), der seit dem Jahr 2000 vom Rechenzentrum der TU Chemnitz betrieben wird. Dieses Clustersystem besteht aus 528 Knoten, die durch ein Fast-Ethernet-Netzwerk mit einer Übertragungsbandbreite von 100 MBit/s verbunden sind. Die Knoten besitzen einen Pentium-III-Prozessor mit einer Taktfrequenz von 800 MHz und sind mit je 512 MB Hauptspeicher (SDRAM PC100) ausgestattet. Mit einer Linpack-Performance von 221,6 GFLOPS erreichte dieses System im November 2001 Platz 137 der Top-500-Liste (Meuer u. a.).

Das zweite verwendete Clustersystem betreibt der Lehrstuhl für Angewandte Informatik 2 der Universität Bayreuth. Dieses System basiert auf 32 SMP-Knoten des Typs MEGWARE Saxonid C2, die mit je zwei Prozessoren des Typs AMD Opteron DP 246 mit einer Taktfrequenz von 2 GHz sowie mit einem Hauptspeicher von 4 GB ausgestattet sind. Tabelle A.5 faßt die wichtigsten Cache-Parameter der verwendeten Opteron-Prozessoren zusammen. Das Clustersystem verfügt über drei Verbindungsnetzwerke mit unterschiedlichen Übertragungsgeschwindigkeiten: ein Fast-Ethernet-Netzwerk (100 Mbit/s), ein Gigabit-Ether-

net-Netzwerk (1 Gbit/s) und ein Infiniband-Netzwerk (10 Gbit/s). Dadurch ist es möglich, den Einfluß der Geschwindigkeit des Verbindungsnetzwerks auf die Skalierbarkeit von Programmen zu untersuchen.

Stufe	Typ	Größe	Assoziativität	Zeilengröße
1	Daten	64 KB	2-fach	64 Byte
1	Instruktionen	64 KB	2-fach	64 Byte
2	Unified	1 MB	16-fach	64 Byte

Tab. A.5: Cache-Parameter des AMD Opteron DP 246. Quelle: AMD (2005).

A.6 Cray T3E-1200

Zwischen 1997 und 2004 wurde am Forschungszentrum Jülich eine Cray T3E-1200 betrieben. Hierbei handelte es sich um einen eng gekoppelten Parallelrechner, auf dem 512 Prozessoren des Typs DEC Alpha 21164 mit einer Taktfrequenz von 600 MHz für die Ausführung von Anwendungen zur Verfügung standen. Tabelle A.6 zeigt die Cache-Parameter der verwendeten Alpha-Prozessoren. Die Prozessoren der Cray T3E-1200 waren durch ein 3D-Torus-Netzwerk mit einer Übertragungsgeschwindigkeit von 500 MB/s miteinander verbunden. Der Speicher war physikalisch verteilt, wobei jedem Prozessor ein lokaler Speicher von 512 MB zugeordnet war, so daß sich eine Gesamtspeicherkapazität von 256 GB ergab. Auf logischer Ebene bilden jedoch alle Speichermodule einen globalen Adreßraum. Der Zugriff auf Speicher eines anderen Prozessors erfolgte mittels spezieller Operationen (`GET` und `PUT`) über einen externen Registersatz (E-Register). Für weitere Informationen siehe z. B. Oed (1996) und Scott (1996).

Stufe	Typ	Größe	Assoziativität	Zeilengröße
1	Daten	8 KB	direkt	32 Byte
1	Instruktionen	8 KB	direkt	32 Byte
2	Unified	96 KB	3-fach	32 oder 64 Byte

Tab. A.6: Cache-Parameter des DEC Alpha 21164. Quelle: Compaq (1998).

A.7 Jülicher Multiprozessor (JUMP)

Unter den Systemen, auf denen im Rahmen dieser Arbeit Laufzeitexperimente durchgeführt wurden, ist der Jülicher Multiprozessor (JUMP) dasjenige mit der höchsten Gesamtrechenleistung. Dieses SMP-Clustersystem wird seit 2003 am Forschungszentrum Jülich betrieben. Seit Installation der letzten Ausbaustufe Anfang 2004 besteht das System aus insgesamt 41 SMP-Knoten des Typs IBM eServer pSeries 690, von denen jeder mit 32 Power4+-Prozessorkerne mit einer Taktfrequenz von 1,7 GHz ausgestattet ist. Jeder Knoten verfügt über 128 GB Hauptspeicher. Innerhalb der SMP-Knoten können die Prozessoren über einen gemeinsamen Adreßraum kommunizieren. Die Verbindung zwischen den Knoten wird über einen sogenannten High-Performance-Switch hergestellt. Insgesamt erreicht das System eine Linpack-Performance von 5,568 TFLOPS und war damit für einige Zeit der schnellste Supercomputer Europas.

Die SMP-Knoten besitzen bezüglich der physikalischen Anordnung der Prozessoren und der Caches einen hierarchischen Aufbau, der das System für Untersuchungen und Optimierungen des Lokalitätsverhaltens von Programmen interessant macht. Jeder Power4+-Chip enthält zwei Prozessorkerne, und je vier Chips sind auf einem Multi-Chip-Modul (MCM) untergebracht. Jeder Prozessorkern besitzt einen eigenen L1-Cache, der Daten (32 KB) und Instruktionen (64 KB) getrennt speichert. Die beiden Prozessorkerne eines Chips teilen sich einen rund 1,5 MB großen L2-Cache. Der Hauptspeicher ist physikalisch auf die einzelnen Power4+-Chips verteilt, was wie auf der Sun Fire 6800 zu ungleichförmigen Speicherzugriffszeiten führt. Zwischen jedem Chip und dem ihm zugeordneten Hauptspeicher befindet sich ein 32 MB großes L3-Cache-Modul. Die 4 L3-Cache-Module, die zum gleichen Multi-Chip-Modul gehören, bilden zusammen einen 128 MB großen L3-Cache. Tabelle A.7 faßt die Cache-Parameter des Systems zusammen.

Stufe	Typ	Größe	Zuordnung	Assoziativität	Zeilengröße
1	Daten	32 KB	Prozessorkern	2-fach	128 Byte
1	Instruktionen	64 KB	Prozessorkern	direkt	128 Byte
2	Unified	1,5 MB	Chip	8-fach	128 Byte
3	Unified	128 MB	Multi-Chip-Modul	8-fach	512 Byte

Tab. A.7: Cache-Hierarchie des IBM eServer pSeries 690 bei Ausstattung mit IBM-Power4+-Prozessoren. Quelle: Tendler u. a. (2002).

B Verwendete Testprobleme

B.1 BRUSS2D: Brusselator mit Diffusion

Bei diesem häufig – u. a. von Burrage (1995), Hairer u. a. (2000) und Hairer u. Wanner (2002) – verwendeten Testproblem handelt es sich um eine zweidimensionale partielle Differentialgleichung mit folgender Gestalt:

$$\frac{\partial u}{\partial t} = 1 + u^2 v - 4.4u + \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \ ,$$

$$\frac{\partial v}{\partial t} = 3.4u - u^2 v + \alpha \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) \ ,$$

welche die chemische Reaktion zweier Substanzen modelliert. Betrachtet wird das Gebiet

$$0 \le x \le 1 \ , \quad 0 \le y \le 1 \quad \text{für} \quad t \ge 0 \ .$$

Durch Diskretisierung mittels Linienmethode über einem $N \times N$-Gitter mit der Gittergröße $1/(N-1)$ und den Neumann-Randbedingungen

$$\frac{\partial u}{\partial n} = 0 \ , \qquad \frac{\partial v}{\partial n} = 0$$

zusammen mit den Anfangsbedingungen

$$u(x, y, 0) = 0.5 + y \ , \qquad v(x, y, 0) = 1 + 5x$$

erhält man ein gewöhnliches Differentialgleichungssystem der Dimension $n = 2N^2$:

$$\frac{dU_{ij}}{dt} = 1 + U_{ij}^2 V_{ij} - 4.4U_{ij} + \alpha(N-1)^2 \left(U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1} - 4U_{i,j} \right) \ ,$$
$$\frac{dV_{ij}}{dt} = 3.4U_{ij} - U_{ij}^2 V_{ij} + \alpha(N-1)^2 \left(V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1} - 4V_{i,j} \right) \ .$$

Es wurden zwei unterschiedliche Linearisierungen für die Gitterpunkte $\{U_{ij}\}_{i,j=1,\ldots,N}$ und $\{V_{ij}\}_{i,j=1,\ldots,N}$ implementiert:

$U_{11}, U_{12}, \ldots U_{NN}, V_{11}, V_{12}, \ldots, V_{NN}$	(zeilenorientiert, BRUSS2D-ROW),
$U_{11}, V_{11}, U_{12}, V_{12}, \ldots, U_{ij}, V_{ij}, \ldots U_{NN}, V_{NN}$	(gemischt-zeilenorientiert, BRUSS2D-MIX).

Die Zugriffsdistanzen für beide Linearisierungen sind beschränkt. Sie betragen N^2 für BRUSS2D-ROW bzw. $2N$ für BRUSS2D-MIX. Die Funktionsauswertungszeiten sind für beide Linearisierungen für alle Komponenten etwa gleich, da jeweils auf eine nahezu konstante, durch den zur Diskretisierung verwendeten 5-Punkt-Stern vorgegebene Anzahl von Komponenten zugegriffen wird. Eine Ausnahme stellen lediglich die Randpunkte dar. Die Erhöhung der Systemgröße durch Verfeinerung des Diskretisierungsgitters ändert die Anzahl der ausgeführten Instruktionen pro Funktionsauswertung nicht signifikant.

Eine Steuerung der Steifheit ist über den Parameter α möglich, der den Einfluß des Diffusionsterms dämpfen oder verstärken kann. In dieser Arbeit verwenden wir entweder $\alpha = 2 \cdot 10^{-3}$ oder $\alpha = 2 \cdot 10^{-4}$, um ein nichtsteifes System zu erzeugen. Auch die Wahl des Diskretisierungsgitters hat Einfluß auf die Steifheit des Systems. Zur Untersuchung des Lokalitätsverhaltens der Lösungsverfahren sind überwiegend

Systeme mit sehr großer Dimension interessant, deren Speicherbedarf in der gleichen Größenordnung liegt wie die Größe der Cache-Stufen der Zielarchitektur. Da die Cachegröße moderner Rechnersysteme z. T. mehrere MB beträgt, werden deshalb in dieser Arbeit vor allem sehr feine Gitter betrachtet, z. B. $N = 250$ bis $N = 1000$. Die Anzahl der Zeitschritte, die ein eingebettetes Runge-Kutta-Verfahren zur Integration über ein vorgegebenes Intervall ausführen muß, wächst für dieses Testproblem jedoch nur etwa linear mit der Systemdimension, so daß die Anwendung eingebetteter Runge-Kutta-Verfahren auch für große Systeme nicht zu einem Rechenzeitproblem führt.

B.2 MEDAKZO: Medizinisches Akzo-Nobel-Problem

Dieses Testproblem aus der Testmenge von Lioen u. de Swart (1999) beschreibt das Eindringen markierter Antikörper in ein durch einen Tumor infiziertes Gewebe. Das mathematische Modell besteht aus zwei eindimensionalen partiellen Differentialgleichungen

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} - kuv \ , \qquad \frac{\partial v}{\partial t} = -kuv \ ,$$

die auf dem Streifen $S_T = \{(x,t) : 0 < x < \infty, 0 < t < T\}$ betrachtet werden. Als Anfangs- und Randbedingungen werden $u(x,0) = 0$ und $v(x,0) = v_0$ für $x > 0$ verwendet, wobei v_0 konstant ist und $u(0,t) = \phi(t)$ für $0 < t < T$ gilt. Die Diskretisierung mittels Linienmethode führt zu einem System steifer ODEs:

$$y'(t) = f(t,y) \ , \quad y(0) = g$$

mit

$$y \in \mathbb{R}^{2N} \ , \qquad 0 \le t \le 20 \ , \qquad g = (0, v_0, 0, v_0, \ldots, 0, v_0)^T \ .$$

Die Funktion f ist gegeben durch

$$f_{2j-1} = \alpha_j \frac{y_{2j+1} - y_{2j-3}}{\Delta\zeta} + \beta_j \frac{y_{2j-3} - 2y_{2j-1} + y_{2j+1}}{(\Delta\zeta)^2} - k\, y_{2j-1} y_{2j} \ ,$$
$$f_{2j} = -k\, y_{2j} y_{2j-1} \ ,$$

mit $j = 1, \ldots, N$ und

$$\alpha_j = \frac{2(j\Delta\zeta - 1)^3}{c^2} \ , \qquad \beta_j = \frac{(j\Delta\zeta - 1)^4}{c^2} \ , \qquad \Delta\zeta = \frac{1}{N} \ ,$$
$$y_{-1}(t) = \phi(t) \ , \qquad y_{2N+1} = y_{2N-1} \ .$$

Die Funktion $\phi(t)$ wird beschrieben durch

$$\phi(t) = \begin{cases} 2 & \text{for } t \in (0,5] \\ 0 & \text{for } t \in (5,20] \end{cases} \ ,$$

was bedeutet, daß f eine Diskontinuität bei $t = 5$ durchläuft.

Geeignete Werte für k, v_0 und c sind 100, 1 und 4. Die Systemgröße beträgt $n = 2N$ und kann durch Wahl der Anzahl der Gitterlinien N verändert werden. Die Funktionsauswertungszeiten der einzelnen Komponenten liegen alle in $O(1)$ und unterscheiden sich nur geringfügig. Das System besitzt eine sehr kleine Zugriffsdistanz von 2. Insbesondere bei Wahl sehr großer Werte von N wird das System sehr steif, da die Anzahl der Zeitschritte exponentiell mit der Systemgröße wächst.

B.3 STARS: N-Körper-Standardproblem für Sternhaufen

Die Lösung dieses nichtsteifen Problems aus der Himmelsmechanik mittels unterschiedlicher Integrationsverfahren wurde für $N = 25$ z. B. von Lecar (1968) und Hussels (1973) untersucht. Grundlage des Differen-

tialgleichungssystems zweiter Ordnung ist die dreidimensionale Bewegungsgleichung

$$\ddot{\mathbf{r}}_i = \sum_{\substack{j=1 \\ j \neq i}}^{N} G \cdot \frac{m_j}{\left|\mathbf{r}_j - \mathbf{r}_i\right|^3} \cdot \left(\mathbf{r}_j - \mathbf{r}_i\right) \quad \text{für} \quad i = 1, \ldots, N \ ,$$

welche die resultierende Beschleunigung $\ddot{\mathbf{r}}_i$ der einzelnen Sterne beschreibt. Für die Berechnung der auf einen Stern wirkenden Beschleunigung muß die Gravitationswirkung aller anderen Sterne berücksichtigt werden, welche die Beschleunigung, d. h. die zweite zeitliche Ableitung des Ortes eines Körpers, als Vektorsumme der Teilbeschleunigungen berechnet, die aus den Wechselwirkungen mit allen anderen Körpern resultieren. Alle Sterne besitzen die gleiche Masse $m_i = m$, die vom Benutzer vorgegeben werden kann. In dieser Arbeit wird in allen Experimenten die Masse $m = \frac{1}{25}$ benutzt. Als Gravitationskonstante wird $G = 1$ verwendet. Zum Zeitpunkt $t = 0$ befinden sich alle Körper in Ruhe, d. h.,

$$\mathbf{v}_i = \dot{\mathbf{r}}_i = \mathbf{0} \quad \text{für} \quad i = 1, \ldots, N \ .$$

Da zur Darstellung der Position der N Körper jeweils 3 Koordinaten benötigt werden, besteht das gewöhnliche Differentialgleichungssystem zweiter Ordnung aus $3N$ Gleichungen. Durch die Transformation in ein System erster Ordnung verdoppelt sich die Systemgröße auf $6N$.

Weil für die Berechnung der auf einen Stern wirkenden Beschleunigung die Gravitationswirkung aller anderen Sterne berücksichtigt werden muß, beträgt die Funktionsauswertungszeit zur Berechnung der zweiten Ableitung $O(N)$, während zur Berechnung der ersten Ableitung nur ein Speicherzugriff erforderlich ist, der in Zeit $O(1)$ erfolgen kann. Die rechte Seite des Differentialgleichungssystem ist deshalb stark irregulär, wobei der Grad der Irregularität mit wachsender Systemgröße steigt. Die Zugriffsdistanz ist nicht beschränkt. Sie kann aufgrund der Notwendigkeit zur Berücksichtigung der Wechselwirkung mit allen restlichen $N - 1$ Sternen auch durch eine Umordnung der Komponenten nicht wesentlich verringert werden.

Es wurden zwei verschiedene Linearisierungen implementiert. STARS-CON speichert die beiden Ableitungen zusammenhängend in aufeinanderfolgenden Speicherblöcken ab, d. h. die Geschwindigkeitskomponenten (v_{ix}, v_{iy}, v_{iz}) und die Beschleunigungskomponenten (a_{ix}, a_{iy}, a_{iz}) werden wie folgt angeordnet:

$$v_{1x}, \ldots, v_{Nx}, v_{1y}, \ldots, v_{Ny}, v_{1z}, \ldots, v_{Nz}, a_{1x}, \ldots, a_{Nx}, a_{1y}, \ldots, a_{Ny}, a_{1z}, \ldots, a_{Nz} \ .$$

STARS-MIX realisiert stattdessen eine Verschachtelung der Geschwindigkeits- und Beschleunigungskomponenten:

$$v_{1x}, a_{1x}, \ldots, v_{Nx}, a_{Nx}, v_{1y}, a_{1y}, \ldots, v_{Ny}, a_{Ny}, v_{1z}, a_{1z}, \ldots, v_{Nz}, a_{Nz} \ .$$

B.4 EMEP: Chemischer Teil des EMEP-MSC-W-Ozon-Modells

Hierbei handelt es sich um den chemischen Teil des EMEP-MSC-W-Ozon-Modells, das ca. 140 Reaktionen beschreibt, an denen 66 Substanzen beteiligt sind (vgl. Lioen u. de Swart 1999). Das resultierende steife ODE-System besteht infolgedessen aus 66 Gleichungen, welche die Konzentration der einzelnen Substanzen in der Anzahl der Moleküle pro cm^3 beschreiben. Betrachtet wird das Problem im Zeitintervall

$$14\,400 \leq t \leq 417\,600 \ ,$$

wobei t in Sekunden gemessen wird. Das Integrationsintervall umfaßt also insgesamt 112 Stunden. Bei Sonnenaufgang und Sonnenuntergang, d. h. zu den Zeitpunkten $t = 3600(\pm 4 + 24i)$ für $i = 1, 2, 3, 4, 5$, verändern sich einige Reaktionsgeschwindigkeiten. Aus diesem Grund durchläuft die rechte Seite des gewöhnlichen Differentialgleichungssystems zu diesen Zeitpunkten Unstetigkeiten. Die Formeln zur Berechnung der einzelnen Komponenten der rechten Seite besitzen eine sehr unterschiedliche Komplexität, wodurch eine stark ungleichförmige Verteilung der Funktionsauswertungskosten entsteht, die dieses Problem sehr interessant für die Untersuchung von Lastbalancierungsverfahren macht.

Literaturverzeichnis

Abella u. a. 2002
ABELLA, J.; GONZÀLES, A.; LLOSA, J.; VERA, X.: Near-Optimal Loop Tiling by Means of Cache Miss Equations and Genetic Algorithms. In: *Proceedings of the 2002 ICPP Workshops*, 2002, S. 568–577

AMD 2005
Advanced Micro Devices, Inc.: *Software Optimization Guide for AMD64 Processors, Revision 3.06*. Advanced Micro Devices, Inc., September 2005. – Publication number 25112

Aitken 1932
AITKEN, A. C.: On interpolation by iteration of proportional parts, without the use of differences. In: *Proc. Edinburgh Math. Soc.* Bd. 3, 1932, S. 56–67

Allen u. Kennedy 2002
ALLEN, R.; KENNEDY, K.: *Optimizing Compilers for Modern Architectures: A Dependence Based Approach*. Morgan Kaufmann, 2002

Austin u. a. 2002
AUSTIN, Todd; LARSON, Eric; ERNST, Dan: SimpleScalar: An Infrastructure for Computer System Modeling. In: *Computer* 35 (2002), Februar, Nr. 2, S. 59–67

Bader u. Deuflhard 1983
BADER, G.; DEUFLHARD, P.: A semi-implcit mid-point rule for stiff systems of ordinary differential equations. In: *Numer. Math.* 41 (1983), S. 373–398

Bampis u. Kononov 2001
BAMPIS, E.; KONONOV, A.: On the Approximability of Scheduling Multiprocessor Tasks with Time-dependent Processor and Time Requirements. In: *Proceedings of the IPDPS'01 Workshops*, 2001

Beaumont u. a. 2002
BEAUMONT, Olivier; BOUDET, Vincent; ROBERT, Yves: A realistic model and an efficient heuristic for scheduling with heterogeneous processors. In: *16th International Parallel and Distributed Processing Symposium (IPDPS '02 (IPPS SPDP))*. Washington - Brussels - Tokyo: IEEE Computer Society Press, April 2002. – ISBN 0–7695–1573–8, S. 88

Behling u. a. 2001
BEHLING, Steve; BELL, Ron; FARRELL, Peter; HOLTHOFF, Holger; O'CONNELL, Frank; WEIRG, Will: *The POWER4 Processor Introduction and Tuning Guide*. Austin, Texas: IBM Corporation, International Technical Support Organization, November 2001. – SG24-7041-000

Behrend 2005
BEHREND, Stefan: *Implementierung eingebetteter Runge-Kutta-Verfahren für SMP-Cluster*, Universität Bayreuth, Bachelor-Thesis, August 2005

Benker 2005
BENKER, Hans: *Differentialgleichungen mit MATHCAD und MATLAB*. 1. Auflage. Berlin: Springer-Verlag, 2005. – ISBN 3–540–23440–3

Berland u. a. 2004
BERLAND, Julien; BOGEY, Christophe; BAILLY, Christophe: Optimized explicit schemes: matching and boundary schemes, and 4th- order Runge-Kutta algorithm. In: *10th AIAA/CEAS Aeroacoustics Conference, 10-12 may, Manchester, UK,*, 2004, S. 1–34. – AIAA Paper 2004-2814

Berland u. a. 2005
BERLAND, Julien; BOGEY, Christophe; BAILLY, Christophe: Low-dissipation and low-dispersion fourth-order Runge-Kutta algorithm. In: *Computers & Fluids* (2005). – Erscheint demnächst.

Berrendorf u. Mohr 2003
BERRENDORF, R.; MOHR, B.: *PCL – The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors (Version 2.2)*. Forschungszentrum Jülich, Januar 2003

Blochinger u. Küchlin 2003
BLOCHINGER, W.; KÜCHLIN, W.: The Design of an API for Strict Multithreading in C++. In: *Euro-Par 2003. Parallel Processing*, Springer-Verlag, August 2003 (LNCS 2790), S. 722–731

Blochinger u. a. 1998
BLOCHINGER, Wolfgang; KÜCHLIN, Wolfgang; WEBER, Andreas: The Distributed Object-Oriented Threads System DOTS. In: FERREIRA, A. (Hrsg.); ROLIM, J. (Hrsg.); SIMON, H. (Hrsg.); TENG, S.-H. (Hrsg.): *Fifth Intl. Symp. on Solving Irregularly Structured Problems in Parallel (IRREGULAR '98)*. Berkeley, CA, U.S.A.: Springer-Verlag, August 1998 (LNCS 1457), S. 206–217

Blumofe u. a. 1995
BLUMOFE, R.; JOERG, C.; KUSZMAUL, B.; LEISERSON, C.; RANDALL, K.; ZHOU, Y.: Cilk: An Efficient Multithreaded Runtime System. In: *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, 1995

Bogacki u. Shampine 1989a
BOGACKI, P.; SHAMPINE, L. F.: An efficient 3(2) pair of Runge-Kutta formulas. In: *Appl. Math. Letters* 2 (1989), S. 1–9

Bogacki u. Shampine 1989b
BOGACKI, P.; SHAMPINE, L. F.: An efficient Runge-Kutta (4,5) pair / Math. Dept., SMU, Dallas. 1989 (89-20). – Forschungsbericht

Brankin u. a. 1993
BRANKIN, R.; GLADWELL, I.; SHAMPINE, L.: RKSUITE: A Suite of Explicit Runge-Kutta Codes. In: *Contributions to Numerical Mathematics*. Singapore: World Scientific, 1993 (WSSIAA 2), S. 41–53

Brankin u. a. 1992
BRANKIN, R. W.; GLADWELL, I.; SHAMPINE, L. F.: RKSUITE: A Suite of Runge-Kutta Codes for the Initial Value Problem of ODEs / Department of Mathematics, Southern Methodist University. Dallas, Texas, USA, 1992 (92-S1). – Softreport

Bulirsch u. Stoer 1966
BULIRSCH, R.; STOER, J.: Numerical treatment of ordinanary differential equations by extrapolation methods. In: *Numer. Math.* 8 (1966), S. 1–13

Burns u. a. 1994
BURNS, Greg; DAOUD, Raja; VAIGL, James: LAM: An Open Cluster Environment for MPI. In: *Proceedings of Supercomputing Symposium*, 1994, S. 379–386

Burrage u. a. 1980
BURRAGE, K.; BUTCHER, J. C.; CHIPMAN, F. H.: An implementation of singly-implicit Runge-Kutta methods. In: *BIT* 20 (1980), Nr. 3, S. 326–340

Burrage u. Suhartanto 1997a
BURRAGE, K.; SUHARTANTO, H.: Parallel iterated methods based on multistep Runge-Kutta methods of Radau type. In: *Advances in Computational Mathematics* 7 (1997), Nr. 1–2, S. 37–57

Burrage u. Suhartanto 1997b
BURRAGE, K.; SUHARTANTO, H.: Parallel iterated methods based on multistep Runge-Kutta methods of Radau type for stiff problems. In: *Advances in Computational Mathematics* 7 (1997), Nr. 1–2, S. 59–77

Burrage 1978
BURRAGE, Kevin: A special family of Runge-Kutta methods for solving stiff differential equations. In: *BIT* 18 (1978), Nr. 1, S. 22–41

Burrage 1995
BURRAGE, Kevin: *Parallel and Sequential Methods for Ordinary Differential Equations*. Oxford Science Publications, 1995

Burrage u. Butcher 1980
BURRAGE, Kevin; BUTCHER, J. C.: Nonlinear stability of a general class of differential equation methods. In: *BIT* 20 (1980), Nr. 2, S. 185–203. – ISSN 0006–3835

Butcher 1963
BUTCHER, John C.: Coefficients for the study of Runge-Kutta integration processes. In: *J. Austral. Math. Soc.* 3 (1963), S. 185–201

Butcher 1964a
BUTCHER, John C.: Implicit Runge-Kutta processes. In: *Math. Comp.* 18 (1964), S. 50–64

Butcher 1964b
BUTCHER, John C.: On Runge-Kutta processes of high order. In: *J. Austral. Math. Soc.* 4 (1964), S. 179–194

Butcher 2001
BUTCHER, John C.: General Linear Methods For Stiff Differential Equations. In: *BIT Num. Math.* 41 (2001), Nr. 2, S. 240–264

Butcher 2003
BUTCHER, John C.: *Numerical methods for ordinary differential equations*. Chichester: John Wiley & Sons, 2003. – ISBN 0–471–96758–0

Butenhof 1997
BUTENHOF, D. R.: *Programming with POSIX Threads*. Addison-Wesley, 1997. – ISBN 0–201–63392–2

Calvo u. a. 2004
CALVO, M.; FRANCO, J. M.; RÁNDEZ, L.: Short note: A new minimum storage Runge-Kutta scheme for computational acoustics. In: *Journal of Computational Physics* 201 (2004), Nr. 1, S. 1–12. – ISSN 0021–9991

Carino u. Banicescu 2005
CARINO, R.; BANICESCU, I.: A Load Balancing Tool for Distributed Parallel Loops. In: *Cluster Computing* 8 (2005), Oktober, Nr. 4

Carriero u. Gelernter 1989
CARRIERO, N.; GELERNTER, D.: How to Write Parallel Programs. In: *ACM Computing Surveys* 21 (1989), Nr. 3, S. 323–357

Ceschino 1961
CESCHINO, F.: Modification de la longueur du pas dans l'intégration numérique par les méthodes à pas liés. In: *Chiffres* 2 (1961), S. 101–106

Ceschino 1962
CESCHINO, F.: Evaluation de l'erreure par pas dans les problèmes differentiélles. In: *Chiffres* 5 (1962), S. 223–229

Ceschino u. Kuntzmann 1963
CESCHINO, F.; KUNTZMANN, J.: *Problèmes Différentiels de Conditions Initiales. Méthodes Numériques.* Paris: Dunod, 1963

Charlesworth 2001
CHARLESWORTH, Alan: The Sun Fireplane System Interconnect. In: *SC2001: High Performance Networking and Computing*, ACM Press and IEEE Computer Society Press, 2001

Chipman 1971
CHIPMAN, F. H.: *A*-Stable Runge-Kutta Processes. In: *BIT* 11 (1971), Nr. 4, S. 384–388. – ISSN 0006–3835

Coleman u. McKinley 1995
COLEMAN, S.; MCKINLEY, K. S.: Tile Size Selection Using Cache Organization and Data Layout. In: *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI'95), SIGPLAN Notices*. La Jolla, CA, Juni 1995, S. 279–290

Compaq 1998
Compaq Computer Corporation: *Alpha 21164 Microprocessor Hardware Reference Manual*. Compaq Computer Corporation, Dezember 1998. – Bestellnummer: EC-QP99C-TE

Cong u. a. 2000
CONG, N. H.; PODHAISKY, H.; WEINER, R.: Performance of explicit pseudo two-step RKN methods on a shared memory computer / Martin-Luther-Universität Halle-Wittenberg, Institut für Numerische Mathematik. 2000 (00-21). – Reports on Numerical Mathematics

Cray 2005
Cray, Inc.: *Cray X1E Datasheet*. Cray, Inc., Januar 2005

Culler u. a. 1999
CULLER, David E.; SINGH, Jaswinder P.; GUPTA, Anoop: *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco: Morgan Kaufmann, 1999

Curtis u. Hirschfelder 1952
CURTIS, C. F.; HIRSCHFELDER, J. O.: Integration of stiff equations. In: *Proc. Nat. Acad. Sci.* 38 (1952), S. 235–243

Dandamudi u. Ayachi 1999
DANDAMUDI, Sivarama P.; AYACHI, Samir: Performance of Hierarchical Processor Scheduling in Shared-Memory Multiprocessor Systems. In: *IEEE Transactions on Computers* 48 (1999), Nr. 11, S. 1202–1213

Decker 2000
DECKER, T.: Virtual Data Space – Load Balancing for Irregular Applications. In: *Parallel Computing* 26 (2000), Dezember, Nr. 13–14, S. 1825–1860

Dekker u. Verwer 1984
DEKKER, K.; VERWER, J. G.: *Stability of Runge-Kutta methods for stiff nonlinear differential equations*. Amsterdam: North-Holland, 1984

Delattre 2004
DELATTRE, Franck; WWW.CPUID.COM (Hrsg.): *The AMD K8 Architecture*. Version: Februar 2004. http://www.cpuid.com/reviews/K8/. – Online–Ressource

Denning 1968
DENNING, P. J.: The Working Set Model for Program Behavior. In: *Communications of the ACM* 11 (1968), Mai, Nr. 5, S. 323–333

Deuflhard 1983
DEUFLHARD, P.: Order and stepsize control in extrapolation methods. In: *Numer. Math.* 41 (1983), S. 399–422

Deuflhard 1985
DEUFLHARD, P.: Recent progress in extrapolation methods for ordinary differential equations. In: *SIAM Rev.* 27 (1985), S. 505–535

Deuflhard u. Nowak 1987
DEUFLHARD, P.; NOWAK, U.: Extrapolation integrators for quasilinear implicit ODEs. In: DEUFLHARD, P. (Hrsg.); ENGQUIST, B. (Hrsg.): *Large Scale Scientific Computing*. Boston, Basel, Stuttgart: Birkhäuser-Verlag, 1987, S. 37–50

Dormand u. Prince 1980
DORMAND, J. R.; PRINCE, P. J.: A family of embedded Runge-Kutta formulae. In: *J. Comput. Appl. Math.* 6 (1980), Nr. 1, S. 19–26

Dormand u. Prince 1989
DORMAND, J. R.; PRINCE, P. J.: Practical Runge-Kutta processes. In: *SIAM Journal on Scientific and Statistical Computing* 10 (1989), Nr. 1, S. 977–989

Drosinos u. a. 2004
DROSINOS, N.; GOUMAS, G.; ATHANASAKI, M.; KOZIRIS, N.: Delivering High Performance to Parallel Applications Using Advanced Scheduling. In: *Parallel Computing: Software Technology, Algorithms, Architectures & Applications – Proceedings of the International Conference ParCo2003, Dresden, Germany* Bd. 13, Elsevier, September 2004, S. 233–240

Earth Simulator Center/JAMSTEC 2004
EARTH SIMULATOR CENTER (Hrsg.); JAMSTEC (Hrsg.): *Earth Simulator*. Broschüre, abrufbar unter http://www.es.jamstec.go.jp, April 2004

Edler u. Hill
EDLER, Jan; HILL, Mark D.: *Dinero IV: Trace-Driven Uniprocessor Cache Simulator*. http://www.cs.wisc.edu/~markhill/DineroIV/. – Online–Ressource, Abruf: 7. Jan. 2006

Ehle 1968
EHLE, B. L.: High Order *A*-Stable Methods for the Numerical Solution of Systems of D.E.'s. In: *BIT* 8 (1968), Nr. 4, S. 276–278. – ISSN 0006–3835

Ehrig u. a. 1998
EHRIG, R.; NOWAK, U.; DEUFLHARD, P.: Massively Parallel Linearly-Implicit Extrapolation Algorithms as a Powerful Tool in Process Simulation. In: D'HOLLANDER, E. H. (Hrsg.); JOUBERT, G. R. (Hrsg.); PETERS, F. J. (Hrsg.); TROTTENBERG, U. (Hrsg.): *Parallel Computing: Fundamentals, Applications and New Directions*. Elsevier, 1998, S. 517–524

El-Rewini u. a. 1995
EL-REWINI, H.; ALI, H. H.; LEWIS, T. G.: Task scheduling in multiprocessing systems. In: *IEEE Computer* 28 (1995), Nr. 12, S. 27–37

England 1969
ENGLAND, R.: Error estimates for Runge-Kutta type solutions to systems of ordinary differential equations. In: *The Computer Journal* 12 (1969), Mai, Nr. 2, S. 166–170

Enright u. a. 1995
ENRIGHT, Wayne H.; HIGHAM, Desmond J.; OWREN, Brynjulf; SHARP, Philip W.: A Survey of the Explicit Runge-Kutta Method / University of Toronto, Department of Computer Science. 1995 (94-291). – Forschungsbericht

Euler 1768
EULER, Leonard: *Institutionum calculi integralis*. Bd. 1. St. Petersburg: Petropoli impensis academiae imperialis scientiarum, 1768. – Neuflage in *Opera Omnia*, Reihe 1, Bd. 11, Birkhäuser-Verlag, 1913

Fehlberg 1964
FEHLBERG, E.: New High-Order Runge-Kutta Formulas with Stepsize Control for Systems of First- and Second-Order Differential Equations. In: *Z. Angew. Math. Mech.* 44 (1964), S. T17–T29. – Sonderheft

Fehlberg 1968
FEHLBERG, E.: Classical fifth-, sixth-, seventh- and eighth order Runge-Kutta formulas with step size control / NASA. 1968 (287). – Forschungsbericht. Auszug veröffentlicht in Computing 4 (1969), S. 93–106

Fehlberg 1969
FEHLBERG, E.: Low-order classical Runge-Kutta formulas with step size control and their application to some heat transfer problems / NASA. 1969 (315). – Forschungsbericht. Auszug veröffentlicht in Computing 6 (1970), S. 61–71

Forrington 1961
FORRINGTON, C. V. D.: Extensions of the predictor-corrector method for the solution of systems of ordinary differential equations. In: *The Computer Journal* 4 (1961), April, Nr. 1, S. 80–85

Frank u. van der Houwen 2001
FRANK, J. E.; VAN DER HOUWEN, P. J.: Parallel iteration of the extended backward differentiation formulas. In: *IMA Journal of Numerical Analysis* 21 (2001), Nr. 1, S. 367–385

Frommer u. Pohl 1995
FROMMER, Andreas; POHL, Bert: A comparison result for multisplittings and waveform relaxation methods. In: *Numerical Linear Algebra with Applications* 2 (1995), Nr. 4, S. 335–346

Gabriel u. a. 2004
GABRIEL, Edgar; FAGG, Graham E.; BOSILCA, George; ANGSKUN, Thara; DONGARRA, Jack J.; SQUYRES, Jeffrey M.; SAHAY, Vishal; KAMBADUR, Prabhanjan; BARRETT, Brian; LUMSDAINE, Andrew; CASTAIN, Ralph H.; DANIEL, David J.; GRAHAM, Richard L.; WOODALL, Timothy S.: Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In: *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, September 2004, S. 97–104

Gander u. a. 2003
GANDER, M. J.; HALPERN, L.; NATAF, F.: Optimal Schwarz Waveform Relaxation for the One Dimensional Wave Equation. In: *SIAM J. Numer. Anal.* 41 (2003), Nr. 5, S. 1643–1681

Gara u. a. 2005
GARA, A.; BLUMRICH, M. A.; CHEN, D.; CHIU, G. L.-T.; COTEUS, P.; GIAMPAPA, M. E.; HARING, R. A.; HEIDELBERGER, P.; HOENICKE, D.; KOPCSAY, G. V.; LIEBSCH, T. A.; OHMACHT, M.; STEINMACHER-BUROW, B. D.; TAKKEN, T.; VRANAS, P.: Overview of the Blue Gene/L system architecture. In: *IBM J. Res. & Dev.* 49 (2005), März/Mai, Nr. 2/3, S. 195–212

Gear 1987
GEAR, C. W.: The potential for parallelism in ordinary differential equations. In: *Computational Mathematics II, Proc. 2nd Int. Conf. Numer. Anal. Appl.*, Boole Press, 1987, S. 33–48

Gear 1988
GEAR, C. W.: Parallel methods for ordinary differential equations. In: *Calcolo* 25 (1988), Nr. 1–2, S. 1–20

Gerkhardt 2005
GERKHARDT, Ekaterina: *Lokalitätsoptimierung iterierter Runge-Kutta-Verfahren für gewöhnliche Differentialgleichungssysteme mit beschränkter Zugriffsdistanz*, Universität Bayreuth, Bachelor-Thesis, November 2005

Ghosh u. a. 1999
GHOSH, S.; MARTONOSI, M.; MALIK, S.: Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21 (1999), Nr. 4, S. 703–746

Gladwell u. a. 1987
GLADWELL, I.; SHAMPINE, L. F.; BRANKIN, R. W.: Automatic selection of the initial step size for an ODE solver. In: *J. Comput. Appl. Math.* 18 (1987), Nr. 2, S. 175–192. – ISSN 0377–0427

Gragg 1965
GRAGG, W. B.: On Extrapolation Algorithms for Ordinary Initial Value Problems. In: *SIAM J. Numer. Anal.* 2 (1965), S. 384–404

Graham u. a. 2006
GRAHAM, Richard L.; WOODALL, Timothy S.; SQUYRES, Jeffrey M.: Open MPI: A Flexible High Performance MPI. In: *Parallel Processing and Applied Mathematics. 6th International Conference, PPAM 2005, Poznan, Poland, September 11-14, 2005 Revised Selected Papers (LNCS 3911)*, 2006, S. 228–239

Gropp u. a. 1996
GROPP, W.; LUSK, E.; DOSS, N.; SKJELLUM, A.: A high-performance, portable implementation of the MPI message passing interface standard. In: *Parallel Computing* 22 (1996), September, Nr. 6, S. 789–828

Gropp 2002
GROPP, William: MPICH2: A New Start for MPI Implementations. In: *Proceedings of the 9th European PVM/MPI Users' Group Meeting*, Springer-Verlag, 2002 (LNCS 2474), S. 7

Gustafsson u. a. 1988
GUSTAFSSON, Kjell; LUNDH, Michael; SÖDERLIND, Gustaf: A PI stepsize control for the numerical solution of ordinary differential equations. In: *BIT* 28 (1988), Februar, Nr. 2, S. 270–287

Hairer u. a. 1987
HAIRER, Ernst; NØRSETT, Syvert P.; WANNER, Gerhard: *Solving Ordinary Differential Equations I: Non-stiff Problems*. 1. Auflage. Berlin: Springer-Verlag, 1987

Hairer u. a. 2000
HAIRER, Ernst; NØRSETT, Syvert P.; WANNER, Gerhard: *Solving Ordinary Differential Equations I: Non-stiff Problems*. 2. überarb. Auflage. Berlin: Springer-Verlag, 2000

Hairer u. Wanner 2002
HAIRER, Ernst; WANNER, Gerhard: *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. 2. überarb. Auflage. Berlin: Springer-Verlag, 2002

Hall u. Higham 1988
HALL, G.; HIGHAM, D. J.: Analysis of stepsize selection schemes for Runge-Kutta codes. In: *IMA Journal of Numerical Analysis* 8 (1988), S. 305–310

Handy 1998
HANDY, J.: *the Cache Memory book*. 2. Auflage. San Diego: Academic Press, Inc., 1998

Harder u. a. 2004
HARDER, Helmut; STELLMACH, Stephan; HANSEN, Ulrich: Magnetohydrodynamic Simulations of the Geodynamo. In: WOLF, Dietrich (Hrsg.); MÜNSTER, Gernot (Hrsg.); KREMER, Manfred (Hrsg.): *NIC Symposium 2004*, John von Neumann Institut for Computing, 2004 (NIC Series 20), S. 389–398

Heinrich-Litan u. a. 1998
HEINRICH-LITAN, L.; FISSGUS, U.; SUTTER, St.; MOLITOR, P.; RAUBER, T.: Modeling the Communication Behavior of Distributed Memory Machines by Genetic Programming. In: *Euro-Par '98: Proceedings of the 4th International Euro-Par Conference on Parallel Processing*. London, UK: Springer-Verlag, 1998. – ISBN 3–540–64952–2, S. 273–278

Hennessy u. Patterson 2003
HENNESSY, J. L.; PATTERSON, D. A.: *Computer Architecture: A Quantitative Approach*. 3. Auflage. Morgan Kaufmann, 2003

Herrod 1998
HERROD, Stephen A.: *Using Complete Machine Simulation to Understand Computer System Behavior*, Stanford University, Diss., Februar 1998

Heun 1900
HEUN, Karl: Neue Methode zur approximativen Integration der Differentialgleichungen einer unabhängigen Veränderlichen. In: *Zeitschr. für Math. und Phys.* 45 (1900), S. 23–38

High Performance Fortran Forum 1997
HIGH PERFORMANCE FORTRAN FORUM: High Performance Fortran Language Specification, Version 2.0 / Center for Research on Parallel Computation, Rice University. Houston, TX, Januar 1997 (CRPC-TR92225). – Forschungsbericht

Higham 1993
HIGHAM, Nicholas J.: The accuracy of floating point summation. In: *SIAM J. Sci. Comp.* 14 (1993), Nr. 4, S. 783–799

Hippold u. Rünger 2006
HIPPOLD, Judith; RÜNGER, Gudula: Task Pool Teams: A Hybrid Programming Environment for Irregular Algorithms on SMP Clusters. In: *Concurrency and Computation: Practice and Experience* (2006). – Zur Veröffentlichung angenommen. – ISSN 1532–0626

Hoffmann 2003
HOFFMANN, Ralf: *Reduzierung des Synchronisations- und Verwaltungsoverheads von Taskpools auf Shared-Memory-Systemen*, Martin-Luther-Universität Halle-Wittenberg, Diplomarbeit, April 2003

Hoffmann u. a. 2004a
HOFFMANN, Ralf; KORCH, Matthias; RAUBER, Thomas: Performance Evaluation of Task Pools Based on Hardware Synchronization. In: *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2004, S. 44

Hoffmann u. a. 2004b
HOFFMANN, Ralf; KORCH, Matthias; RAUBER, Thomas: Using Hardware Operations to Reduce the Synchronization Overhead of Task Pools. In: *Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)*. Montreal, Quebec, Canada, August 2004, S. 241–249

Hull u. a. 1976
HULL, T. E.; ENRIGHT, W. H.; JACKSON, K. R.: User's guide for DVERK - A subroutine for solving non-stiff ODEs / Department of Computer Science, University of Toronto. 1976 (100). – Forschungsbericht

Hunold u. Rauber 2005
HUNOLD, Sascha; RAUBER, Thomas: Reducing the Overhead of Intra-Node Communication in Clusters of SMPs. In: *Proceedings of the 3rd International Symposium on Parallel and Distributed Processing and Applications (ISPA 2005)*. Nanjing, China: Springer-Verlag, November 2005

Hussels 1973
HUSSELS, Hans-Günter: *Schrittweitensteuerung bei der Integration gewöhnlicher Differentialgleichungen mit Extrapolationsverfahren*. Köln, Deutschland, Universität zu Köln, Diplomarbeit, 1973

Hwang u. Saltz 2003
HWANG, Y.-S.; SALTZ, J.: Identifying Parallelism in Programs with Cyclic Graphs. In: *Journal of Parallel and Distributed Computing* 63 (2003), Nr. 3, S. 337–355

IMSL-Homepage
VISUAL NUMERICS, INC. (Hrsg.): *IMSL Numerical Libraries Family of Products*. `http://www.vni.com/products/imsl/`. – Online–Ressource, Abruf: 12. Dez. 2005

Intel 2002
Intel Corporation: *Intel Itanium 2 Processor Hardware Developer's Manual*. Intel Corporation, Juli 2002. – Document number 2511109-001

Intel 2004
Intel Corporation: *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*. Intel Corporation, Mai 2004. – Document number 251110-003

Intel 2006a
Intel Corporation: *IA-32 Intel Architecture Optimization Reference Manual*. Intel Corporation, April 2006. – Order number 248966-013US

Intel 2006b
Intel Corporation: *IA-32 Intel Architecture Software Developer's Manual, Volumes 1–4*. Intel Corporation, Juni 2006. – Document numbers 253665–253669

Irigoin u. Triolet 1988
IRIGOIN, F.; TRIOLET, R.: Supernode Partitioning. In: *15th Annual ACM Symposium on Principles of Programming Languages*. San Diego, Calif., Januar 1988, S. 319–329

Iserles u. Nørsett 1990
ISERLES, A.; NØRSETT, S. P.: On the theory of parallel Runge-Kutta methods. In: *IMA Journal of Numerical Analysis* 10 (1990), Nr. 4, S. 463–488. – ISSN 0272–4979

Jackson u. Nørsett 1995
JACKSON, K. R.; NØRSETT, S. P.: The Potential for Parallelism in Runge-Kutta Methods. Part 1: RK Formulas in Standard Form. In: *SIAM J. Numer. Anal.* 32 (1995), Februar, Nr. 1, S. 49–82. – ISSN 0036–1429 (print), 1095–7170 (electronic)

Jeltsch u. Pohl 1991
JELTSCH, Rolf; POHL, Bert: Waveform relaxation with overlapping splittings / ETH, Seminar für Angewandte Mathematik. Zürich, Schweiz, 1991. – Forschungsbericht

Jin u. a. 2001
JIN, G.; MELLOR-CRUMMEY, J.; FOWLER, R.: Increasing Temporal Locality with Skewing and Recursive Blocking. In: *SC2001: High Performance Networking and Computing*, ACM Press and IEEE Computer Society Press, November 2001. – CD-ROM

Kahle u. a. 2005
KAHLE, J. A.; DAY, M. N.; HOFSTEE, H. P.; JOHNS, C. R.; MAEURER, T. R.; SHIPPY, D.: Introduction to the Cell multiprocessor. In: *IBM J. Res. & Dev.* 49 (2005), Juli/September, Nr. 4/5, S. 589–604

Kale u. Krishnan 1996
KALE, Laxmikant V.; KRISHNAN, Sanjeev: CHARM++. In: WILSON, Gregory V. (Hrsg.); LU, Paul (Hrsg.): *Parallel Programming in C++*. Cambridge, MA: MIT Press, 1996, Kapitel 5, S. 175–214

Kalla u. a. 2003
KALLA, Ron; SINHAROY, Balaram; TENDLER, Joel: Simultaneous Multi-threading Implementation in POWER5 – IBM's Next Generation POWER Microprocessor / IBM Systems Group. 2003. – Forschungsbericht

Kandemir u. a. 2000a
KANDEMIR, M.; CHOUDHARY, A.; RAMANUJAM, J.; KANDASWAMY, M. A.: A Unified Framework for Optimizing Locality, Parallelism, and Communication in Out-of-Core Computations. In: *IEEE Transactions on Parallel and Distributed Systems* 11 (2000), Juli, Nr. 7, S. 648–668. – ISSN 1045–9219

Kandemir u. a. 2000b
KANDEMIR, M.; RAMANUJAM, J.; CHOUDHARY, A.: Compiler algorithms for optimizing locality and parallelism on shared and distributed memory machines. In: *Journal of Parallel and Distributed Computing* 60 (2000), August, S. 924–965

Kappeller u. a. 1996
KAPPELLER, M.; KIEHL, M.; PERZL, M.; LENKE, M.: Optimized Extrapolation Methods for Parallel Solution of IVPs on Different Computer Architectures. In: *Applied Mathematics and Computation 77* (1996), Juli, Nr. 2–3, S. 301–315. – ISSN 0096–3003

Kelm u. Rettig 2004
KELM, Peter; RETTIG, Frank: Analyse transversaler Riemenschwingungen bei Pkw-Aggregatetrieben. In: *Motortechnische Zeitschrift (MTZ)* (2004), Nr. 7, S. 576–581

Kennedy u. Carpenter 1994
KENNEDY, Christopher A.; CARPENTER, Mark H.: Third-order 2N-storage Runge-Kutta schemes with error control / National Aeronautics and Space Administration. Langley Research Center, Hampton, VA, 1994 (NASA TM-109111). – Forschungsbericht

Kennedy u. a. 2000
KENNEDY, Christopher A.; CARPENTER, Mark H.; LEWIS, R. M.: Low-storage, explicit Runge-Kutta schemes for the compressible Navier-Stokes equations. In: *Appl. Numer. Math.* 35 (2000), Nr. 3, S. 177–219

Khalaf u. Hutchinson 1992
KHALAF, B. M. S.; HUTCHINSON, D.: Parallel Algorithms for Initial Value Problems: Parallel Shooting. In: *Parallel Computing* 18 (1992), S. 661–673

Kitagawa u. a. 2003
KITAGAWA, Kenji; TAGAYA, Satoru; HAGIHARA, Yasuhiko; KANOH, Yasushi: A Hardware Overview of SX-6 and SX-7 Supercomputer. In: *NEC Res. & Develop.* 44 (2003), Januar, Nr. 1, S. 2–7

Kleber 2001
KLEBER, A.: Simulation of Air Flow Around an OPEL ASTRA Vehicle with FLUENT / Fluent, Inc. 2001 (JA132). – Journal Articles By Fluent Software Users

Kongetira u. a. 2005
KONGETIRA, Poonacha; AINGARAN, Kathirgamar; OLUKOTUN, Kunle: Niagara: A 32-Way Multithreaded SPARC Processor. In: *IEEE Micro* 25 (2005), März/April, Nr. 2, S. 21–29

Korch u. Rauber 2002
KORCH, Matthias; RAUBER, Thomas: Evaluation of Task Pools for the Implementation of Parallel Irregular Algorithms. In: *Proceedings of the 2002 ICPP Workshops*, IEEE Computer Society Press, August 2002, S. 597–604

Korch u. Rauber 2004a
KORCH, Matthias; RAUBER, Thomas: Comparison of Parallel Implementations of Runge-Kutta Solvers: Message Passing vs. Threads. In: *Parallel Computing: Software Technology, Algorithms, Architectures & Applications – Proceedings of the International Conference ParCo2003, Dresden, Germany* Bd. 13. Elsevier, September 2004, S. 209–216

Korch u. Rauber 2004b
KORCH, Matthias; RAUBER, Thomas: A comparison of task pools for dynamic load balancing of irregular algorithms. In: *Concurrency and Computation: Practice and Experience* 16 (2004), Januar, S. 1–47

Kowarschik u. a. 2002
KOWARSCHIK, M.; RÜDE, U.; THÜREY, N.; WEISS, C.: Performance Optimization of 3D Multigrid on Hierarchical Memory Architectures. In: *Proceedings of the 2002 Conference on Applied Parallel Computing (PARA'02)*. Espoo, Finnland: Springer-Verlag, Juni 2002 (LNCS 2367), S. 307–318

Kowarschik u. Weiß 2003
KOWARSCHIK, M.; WEISS, C.: An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In: *Algorithms for Memory Hierarchies*. Springer-Verlag, 2003 (LNCS 2625)

Krogh 1969
KROGH, Fred T.: A variable step variable order multistep method for the numerical solution of ordinary differential equations. In: MORRELL, A. J. H. (Hrsg.): *Information Processing 68: Proc. IFIP Congress, Edinburgh, UK, 1968* Bd. 1. Amsterdam: North-Holland, 1969, S. 194–199

Kumar u. a. 1994
KUMAR, V.; GRAMA, A. Y.; VEMPATY, N. R.: Scalable Load Balancing Techniques for Parallel Computers. In: *Journal of Parallel and Distributed Computing* 22 (1994), Nr. 1, S. 60–79

Kuntzmann 1961
KUNTZMANN, J.: Neuere Entwickelungen der Methode von Runge-Kutta. In: *Z. Angew. Math. Mech.* 41 (1961), S. 28–31

Kutta 1901
KUTTA, Wilhelm M.: Beitrag zur näherungsweisen Integration totaler Differentialgleichungen. In: *Zeitschr. für Math. und Phys.* 46 (1901), S. 435–453

LaMarca u. Ladner 1997
LAMARCA, A.; LADNER, R. E.: The Influence of Caches on the Performance of Sorting. In: *Proc. 8th ACM Symp. Discrete Algorithms*, 1997, S. 370–379

Lecar 1968
LECAR, M.: Comparison of eleven numerical integrations of the same gravitational 25-body problem. In: *Bulletin Astronomique* 3 (1968), S. 91

Lelarasmee 1982
LELARASMEE, E.: *The waveform relaxation method for the time-domain analysis of large scale nonlinear dynamical systems*. Berkeley, U.S.A., University of California, Diss., 1982

Lelarasmee u. a. 1982
LELARASMEE, E.; RUEHLI, A. E.; SANGIOVANNI-VINCENTELLI, A. L.: The Waveform Relaxation Method for Time-Domain Analysis of Large Scale Integrated Circuits. In: *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 1 (1982), Juli, S. 131–145

Lepère u. a. 2002
LEPÈRE, R.; TRYSTRAM, D.; WOEGINGER, G. J.: Approximation Algorithms for Scheduling Malleable Tasks under Precedence Constraints. In: *Int. J. Found. Comp. Sci.* 13 (2002), Nr. 4, S. 613

Lindenmaier u. a. 2000
LINDENMAIER, Götz; MCKINLEY, Kathryn S.; TEMAM, Olivier: Load Scheduling with Profile Information. In: *Europar 2000. Parallel Processing* Bd. 1900, 2000, S. 223–233

Lioen u. de Swart 1999
LIOEN, W. M.; DE SWART, J. J. B.: *Test Set for Initial Value Problem Solvers, Release 2.1*. Amsterdam, The Netherlands: CWI, September 1999

Lubich 1992
LUBICH, Ch.: Chebyshev acceleration of Picard-Lindelöf iteration. In: *BIT* 32 (1992), Nr. 3, S. 535–538. – ISSN 0006–3835

Lubich u. Ostermann 1987
LUBICH, Ch.; OSTERMANN, A.: Multigrid dynamic iteration for parabolic equations. In: *BIT* 27 (1987), Nr. 2, S. 216–234. – ISSN 0006–3835

Magnusson u. a. 2002
MAGNUSSON, P. S.; CHRISTENSSON, M.; ESKILSON, J.; FORSGREN, D.; HÅLLBERG, G.; HÖGBERG, J.; LARSSON, F.; MOESTEDT, A.; WERNER, B.: Simics: A Full System Simulation Platform. In: *Computer* 35 (2002), Februar, Nr. 2, S. 50–58. – ISSN 0018–9162

Majewski u. a. 2003
MAJEWSKI, Detlev; SCHÄTTLER, Ulrich; DAMRATH, Ulrich: Elektronische Wetterfrösche – Wie Computer das Wetter vorhersagen. In: *c't Magazin für Computer und Technik* (2003), Nr. 15, S. 134

Marr u. a. 2002
MARR, Deborah T.; BINNS, Frank; HILL, David L.; HINTON, Glenn; KOUFATY, David A.; MILLER, J. A.; UPTON, Michael: Hyper-Threading Technology Architecture and Microarchitecture. In: *Intel Technology Journal* 6 (2002), Februar, Nr. 1, S. 4–15. – ISSN 1535–766X

Mathis u. a. 2005
MATHIS, H. M.; MERICAS, A. E.; MCCALPIN, J. D.; EICKEMEYER, R. J.; KUNKEL, S. R.: Characterization of simultaneous multithreading (SMT) efficiency in POWER5. In: *IBM J. Res. & Dev.* 49 (2005), Juli/September, Nr. 4/5, S. 555–564

MATLAB-Homepage
THE MATHWORKS, INC. (Hrsg.): *MATLAB – Die Sprache für wissenschaftlich-technische Berechnungen*. http://www.mathworks.de/products/matlab/. – Online–Ressource, Abruf: 12. Dez. 2005

McKinley 1998
MCKINLEY, K. S.: A Compiler Optimization Algorithm for Shared-Memory Multiprocessors. In: *IEEE Transactions on Parallel and Distributed Systems* 9 (1998), August, Nr. 8, S. 769–787. – ISSN 1045–9219

Merson 1957
MERSON, R. H.: An operational method for the study of integration processes. In: *Proc. Symp. Data Processing*. Salisbury, South Australia: Weapons Research Establishment, 1957, S. 110–1 bis 110–25

Meuer u. a.
MEUER, Hans; STROHMAIER, Erich; DONGARRA, Jack; SIMON, Horst D.: *Top500 Supercomputer Sites*. http://www.top500.org/. – Online–Ressource, Abruf: Februar 2006

Meyer u. a. 2003
MEYER, Ulrich (Hrsg.); SANDERS, Peter (Hrsg.); SIBEYN, Jop (Hrsg.): *LNCS*. Bd. 2625: *Algorithms for Memory Hierarchies*. Berlin: Springer-Verlag, 2003

Miranker u. Liniger 1967
MIRANKER, W. L.; LINIGER, W.: Parallel Methods for the Numerical Integration of Ordinary Differential Equations. In: *Mathematics of Computation* 21 (1967), Nr. 99, S. 303–320

MPI Forum 1994
MPI FORUM: *MPI: A Message-Passing Interface Standard, Version 1.0*. University of Tennessee, Mai 1994. http://www.mpi-forum.org

MPI Forum 1995
MPI FORUM: *MPI: A Message-Passing Interface Standard, Version 1.1*. University of Tennessee, Juni 1995. http://www.mpi-forum.org

MPI Forum 1997
MPI FORUM: *MPI-2: Extensions to the Message-Passing Interface*. University of Tennessee, Juli 1997. http://www.mpi-forum.org

Muchnick 1997
MUCHNICK, S.: *Advanced Compiler Design Implementation*. Morgan Kaufmann, 1997

NAG-Webseite Numerikkomponenten
GROUP, The Numerical A. (Hrsg.): *NAG Numerical Components*. http://www.nag.co.uk/numeric/. – Online–Ressource, Abruf: 12. Dez. 2005

Nevanlinna 1990
NEVANLINNA, Olavi: Linear acceleration of Picard-Lindelöf iteration. In: *Numer. Math.* 57 (1990), April, Nr. 2, S. 147–156. – ISSN 0029–599X (print), 0945–3245 (electronic)

Neville 1934
NEVILLE, E. H.: Iterative interpolation. In: *Ind. Math. Soc. J.* 20 (1934), S. 87–120

NIC-Broschüre 2005
NIC-DIREKTORIUM (Hrsg.): *John von Neumann-Institut für Computing*. Broschüre, abrufbar unter http://www.fz-juelich.de/nic/Publikationen/, 2005

Nieh u. Levoy 1992
NIEH, Jason; LEVOY, Marc: Volume Rendering on Scalable Shared-Memory MIMD Architectures. In: *Proceedings of the Boston Workshop on Volume Visualization*, ACM Press, Oktober 1992, S. 17–24

Nordsieck 1962
NORDSIECK, Arnold: On Numerical Integration of Ordinary Differential Equations. In: *Math. Comp.* 16 (1962), Januar, Nr. 77, S. 22–49. – ISSN 0025–5718

Norman u. Thanisch 1993
NORMAN, M. G.; THANISCH, P.: Models of Machines and Computation for Mapping in Multicomputers. In: *ACM Computing Surveys* 25 (1993), Nr. 3, S. 263–302

Nørsett u. Simonsen 1989
NØRSETT, S. P.; SIMONSEN, H. H.: Aspects of Parallel Runge-Kutta methods. In: *Numerical Methods for Ordinary Differential Equations*, 1989 (LNM 1386), S. 103–117

Oed 1996
OED, Wilfired: *Massiv-paralleles Rechnersystem CRAY T3E*. Cray Research GmbH, März 1996

OpenMP API 2.5 2005
OpenMP Architecture Review Board: *OpenMP Application Program Interface Version 2.5*. OpenMP Architecture Review Board, Mai 2005

Petcu 1999
PETCU, Dana: Solving Initial Value Problems with a Multiprocessor Code. In: *PaCT '99: Proceedings of the 5th International Conference on Parallel Computing Technologies*. London, UK: Springer-Verlag, 1999 (LNCS 1662). – ISBN 3–540–66363–0, S. 452–465

Petcu u. Dragan 2000
PETCU, Dana; DRAGAN, Mircea: Designing an ODE solving environment. In: LANGTANGEN, H. P. (Hrsg.); BRUASET, A. M. (Hrsg.); QUAK, E. (Hrsg.): *Advances in Software Tools for Scientic Computing*. Berlin: Springer-Verlag, 2000 (LNCSE 10), S. 319–338

Podhaisky u. a. 1999
PODHAISKY, H.; WEINER, R.; WENSCH, J.: High Order Explicit Two-Step Runge-Kutta Methods for Parallel Computers / Martin-Luther-Universität Halle-Wittenberg, Institut für Numerische Mathematik. 1999 (99-19). – Reports on Numerical Mathematics

Pohl 1992
POHL, Bert: *Ein Algorithmus zur Lösung von Anfangswertproblemen auf Parallelrechnern*. Zürich, Schweiz, Seminar für Angewandte Mathematik, ETH, Diss., 1992

Polychronopoulos 1988
POLYCHRONOPOULOS, C. D.: *Parallel Programming and Compilers*. Kluwer Academic Publishers, 1988

Polychronopoulos u. Kuck 1987
POLYCHRONOPOULOS, C. D.; KUCK, D. J.: Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. In: *IEEE Transactions on Computers* C-36 (1987), Dezember, Nr. 12, S. 1425–1439

Prince u. Dormand 1981
PRINCE, P. J.; DORMAND, J. R.: High order embedded Runge-Kutta formulae. In: *J. Comput. Appl. Math.* 7 (1981), Nr. 1, S. 67–75

Randall 1998
RANDALL, K. H.: *Cilk: Efficient Multithreaded Computing*, MIT Department of Electrical Engineering and Computer Science, Diss., Juni 1998

Rauber u. Rünger 2004
RAUBER, T.; RÜNGER, G.: Improving Locality for ODE Solvers by Program Transformations. In: *Scientific Programming* 12 (2004), Nr. 3, S. 133–154

Rauber u. Rünger 1996
RAUBER, Thomas; RÜNGER, Gudula: Parallel implementations of iterated Runge-Kutta methods. In: *International Journal of Supercomputer Applications* 10 (1996), Nr. 1, S. 62–90

Rauber u. Rünger 1997a
RAUBER, Thomas; RÜNGER, Gudula: Load Balancing Schemes for Extrapolation Methods. In: *Concurrency: Practice and Experience* 9 (1997), Nr. 3, S. 181–202

Rauber u. Rünger 1997b
RAUBER, Thomas; RÜNGER, Gudula: PVM and MPI Communication Operations on the IBM SP2: Modeling and Comparison. In: *Proc. 11th Symp. on High Performance Computing Systems (HPCS'97)*, 1997, S. 141–152

Rauber u. Rünger 1999a
RAUBER, Thomas; RÜNGER, Gudula: Diagonal-Implicitly Iterated Runge-Kutta Methods on Distributed Memory Machines. In: *International Journal of High Speed Computing* 10 (1999), Nr. 2, S. 185–207

Rauber u. Rünger 1999b
RAUBER, Thomas; RÜNGER, Gudula: Parallel Execution of Embedded and Iterated Runge-Kutta Methods. In: *Concurrency: Practice and Experience* 11 (1999), Nr. 7, S. 367–385

Rauber u. Rünger 1999c
RAUBER, Thomas; RÜNGER, Gudula: Parallel Solution of Stiff Ordinary Differential Equations. In: YANG, T. (Hrsg.): *Parallel Numerical Computations with Applications*, Kluwer Academic Publishers, 1999, S. 33–51

Rauber u. Rünger 2000
RAUBER, Thomas; RÜNGER, Gudula: *Parallele und verteilte Programmierung*. Berlin: Springer-Verlag, 2000

Rauber u. Rünger 2001
RAUBER, Thomas; RÜNGER, Gudula: Optimizing Locality for ODE Solvers. In: *Proceedings of the 15th ACM International Conference on Supercomputing*, ACM Press, 2001, S. 123–132

Richardson 1910
RICHARDSON, L. F.: The approximate arithmetical solution by finite differences of physical problems including differential equations, with an application to the stresses in a mansory dam. In: *Phil. Trans. A* 210 (1910), S. 307–357

Richardson 1927
RICHARDSON, L. F.: The deferred approach to the limit. In: *Phil. Trans. A* 226 (1927), S. 299–349

Romberg 1955
ROMBERG, W.: Vereinfachte numerische Integration. In: *Norske Vid. Selsk. Forhdl.* 28 (1955), S. 30–36

Rosenbrock 1963
ROSENBROCK, H. H.: Some general implicit processes for the numerical solution of differential equations. In: *The Computer Journal* 5 (1963), S. 329–330

Runge 1895
RUNGE, Carl: Ueber die numerische Auflösung von Differentialgleichungen. In: *Mathematische Annalen* 46 (1895), S. 167–178

Runge 1905
RUNGE, Carl: Ueber die numerische Auflösung totaler Differentialgleichungen. In: *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse* (1905), S. 252–257

Ruuth 2005
RUUTH, Steven J.: Global optimization of explicit strong-stability-preserving Runge-Kutta methods. In: *Math. Comp.* 75 (2005), Nr. 253, S. 183–207

Sarafyan 1966
SARAFYAN, D.: Error estimation for Runge-Kutta methods through pseudo-iterative formulas / Louisiana State University. New Orleans, Mai 1966 (14). – Forschungsbericht

Schaller 1998
SCHALLER, C.: *Eine effiziente Implementierung eines Waveform-Relaxations-Verfahrens auf parallelen Rechnerarchitekturen*. Düsseldorf: VDI-Verlag, 1998. – ISBN 3–18–353310–3

Schmitt u. Weiner 2004
SCHMITT, Bernhard A.; WEINER, Rüdiger: Parallel Two-Step W-Methods with Peer Variables. In: *SIAM J. Numer. Anal.* 42 (2004), Nr. 1, S. 265–282

Scott 1996
SCOTT, Steven L.: Synchronization and Communication in the T3E Multiprocessor. In: *ASPLOS-VII, Cambridge, MA, October 2–4, 1996*, 1996

SGI 2003
The SGI Altix 3000 Global Shared-Memory Architecture / Silicon Graphics, Inc. 2003. – Whitepaper

SGI 2005
SGI NUMAlink: Industry Leading Interconnect Technology / Silicon Graphics, Inc. 2005. – Whitepaper

Shampine u. Gladwell 1996
SHAMPINE, L. F.; GLADWELL, I.: Software based on explicit RK formulas. In: *Appl. Numer. Math.* 22 (1996), Nr. 1–3, S. 293–308

Shampine u. Thompson 2000
SHAMPINE, L. F.; THOMPSON, S.: Event Location for Ordinary Differential Equations. In: *Comp. & Maths. with Appls.* 39 (2000), S. 43–54

Shampine u. a. 1976
SHAMPINE, L. F.; WATTS, H. A.; DAVENPORT, S. M.: Solving nonstiff ordinary differential equations – The state of the art. In: *SIAM Rev.* 18 (1976), Juli, Nr. 3, S. 376–410

Shampine u. Reichelt 1997
SHAMPINE, Lawrence F.; REICHELT, Mark W.: The MATLAB ODE Suite. In: *SIAM J. Sci. Comp.* 18 (1997), Nr. 1, S. 1–22

Sinharoy u. a. 2005
SINHAROY, B.; KALLA, R. N.; TENDLER, J. M.; EICKEMEYER, R. J.; JOYNER, J. B.: POWER5 system microarchitecture. In: *IBM J. Res. & Dev.* 49 (2005), Juli/September, Nr. 4/5, S. 505–521

Skeel 1989
SKEEL, Robert D.: Waveform iteration and the shifted Picard splitting. In: *SIAM Journal on Scientific and Statistical Computing* 10 (1989), Juli, Nr. 4, S. 756–776. – ISSN 0196–5204

Squyres u. Lumsdaine 2003
SQUYRES, Jeffrey M.; LUMSDAINE, Andrew: A Component Architecture for LAM/MPI. In: *Proceedings, 10th European PVM/MPI Users' Group Meeting*. Venice, Italy: Springer-Verlag, September/Oktober 2003 (LNCS 2840), S. 379–387

Steihaug u. Wolfbrandt 1979
STEIHAUG, Trond; WOLFBRANDT, Arne: An attempt to avoid exact Jacobian and nonlinear equations in the numerical solution of stiff differential equations. In: *Math. Comp.* 33 (1979), April, Nr. 146, S. 521–534. – ISSN 0025–5718

Strehmel u. Weiner 1982
STREHMEL, K.; WEINER, R.: Behandlung steifer Anfangswertprobleme gewöhnlicher Differentialgleichungen mit adaptiven Runge-Kutta-Methoden. In: *Computing* 29 (1982), Nr. 2, S. 153–165

Strehmel u. Weiner 1995
STREHMEL, Karl; WEINER, Rüdiger: *Numerik gewöhnlicher Differentialgleichungen*. Stuttgart: Teubner-Verlag, 1995

Sun 2003a
An Overview of UltraSPARC III Cu: UltraSPARC III Moves to Copper Technology, Version 1.1 / Sun Microsystems, Inc. 2003 (1-800-555-9SUN). – Whitepaper

Sun 2003b
Solaris Memory Placement Optimization and Sun Fire Servers / Sun Microsystems, Inc. 2003. – Technical Whitepaper

Sun 2004
UltraSPARC IV Processor Architecture Overview / Sun Microsystems, Inc. 2004. – Whitepaper

Supercomputing Technologies Group 2001
SUPERCOMPUTING TECHNOLOGIES GROUP: *Cilk-5.3.2 Reference Manual*. Cambridge: MIT Laboratory for Computer Science, November 2001

Söderlind 2002
SÖDERLIND, Gustaf: Automatic control and adaptive time-stepping. In: *Numerical Algorithms* 31 (2002), S. 281–310

Söderlind 2003
SÖDERLIND, Gustaf: Digital filters in adaptive time-stepping. In: *ACM Trans. Math. Softw. (TOMS)* 29 (2003), Nr. 1, S. 1–26. – ISSN 0098–3500

Söderlind u. Wang 2006
SÖDERLIND, Gustaf; WANG, Lina: Adaptive time-stepping and computational stability. In: *J. Comput. Appl. Math.* 185 (2006), Januar, Nr. 2, S. 225–243

Tabirca u. a. 2004
TABIRCA, S.; TABIRCA, T.; YANG, L. T.; FREEMAN, L: Evaluation of the Feedback Guided Dynamic Loop Scheduling (FGDLS) Algorithms. In: *IEICE Trans. Inf. & Syst.* (2004). – ISSN 0916–8532

Tendler u. a. 2002
TENDLER, J. M.; DODSON, J. S.; FIELDS, JR., J. S.; LEE, H.; SINHAROY, B.: POWER4 system microarchitecture. In: *IBM J. Res. & Dev.* 46 (2002), Januar, Nr. 1

Tullsen u. a. 1995
TULLSEN, Dean M.; EGGERS, Susan; LEVY, Henry M.: Simultaneous Multithreading: Maximizing On-Chip Parallelism. In: *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995, S. 392–403

Turek u. a. 1992
TUREK, J.; WOLF, J.; YU, P.: Approximate algorithms for scheduling parallelizable tasks. In: *ACM Symposium on Parallel Algorithms and Architectures*, 1992, S. 323–332

van der Houwen u. de Swart 1997
VAN DER HOUWEN, P. J.; DE SWART, J. J. B.: Triangularly Implicit Iteration Methods for ODE-IVP Solvers. In: *SIAM J. Sci. Comp.* 18 (1997), Nr. 1, S. 41–55

van der Houwen u. Messina 1999
VAN DER HOUWEN, P. J.; MESSINA, E.: Parallel Adams methods. In: *J. Comput. Appl. Math.* 101 (1999), Januar, S. 153–165

van der Houwen u. Sommeijer 1990a
VAN DER HOUWEN, P. J.; SOMMEIJER, B. P.: Parallel iteration of high-order Runge-Kutta Methods with stepsize control. In: *J. Comput. Appl. Math.* 29 (1990), S. 111–127

van der Houwen u. Sommeijer 1990b
VAN DER HOUWEN, P. J.; SOMMEIJER, B. P.: Parallel ODE Solvers. In: *Proceedings of the ACM International Conference on Supercomputing*, 1990, S. 71–81

van der Houwen u. a. 1992
VAN DER HOUWEN, P. J.; SOMMEIJER, B. P.; CONG, N. H.: Parallel diagonally implicit Runge–Kutta–Nyström methods. In: *Appl. Numer. Math.* 9 (1992), Februar, Nr. 2, S. 111–131. – ISSN 0168–9274

van Lent u. Vandewalle 2002
VAN LENT, J.; VANDEWALLE, S.: Multigrid Waveform Relaxation for Anisotropic Partial Differential Equations. In: *Numerical Algorithms* 31 (2002), Dezember, Nr. 1–4, S. 361–380

Vandewalle 1993
VANDEWALLE, S.: *Parallel Multigrid Waveform Relaxation for Parabolic Problems*. Stuttgart: Teubner-Verlag, 1993

Verner 1978
VERNER, J. H.: Explicit Runge-Kutta methods with estimates of the local truncation error. In: *SIAM J. Numer. Anal.* 15 (1978), S. 772–790

Wallin u. a. 2003
WALLIN, D.; JOHANSSON, H.; HOLMGREN, S.: Cache Memory Behavior of Advanced PDE Solvers / Department of Information Technology, Uppsala University. 2003 (2003-044). – Forschungsbericht

Wanner 1977
WANNER, G.: On the integration of stiff differential equations. In: DESCLOUX, J. (Hrsg.); MARTI, J. (Hrsg.): *Numerical Analysis* Bd. 37. Basel: Birkhäuser-Verlag, 1977, S. 209–226

Weiner u. a. 2003
WEINER, R.; SCHMITT, B. A.; PODHAISKY, H.: Two-step W-Methods and their application to MOL-systems / Martin-Luther-Universität Halle-Wittenberg, Institut für Numerische Mathematik. 2003 (03-02). – Reports on Numerical Mathematics

Wilke u. a. 2003
WILKE, J.; POHL, T.; KOWARSCHIK, M.; RÜDE, U.: Cache Performance Optimizations for Parallel Lattice Boltzmann Codes. In: *Euro-Par 2003. Parallel Processing (LNCS 2790)*. Klagenfurt, Austria: Springer-Verlag, August 2003

Wolfe 1996
WOLFE, M.: *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996

Woo u. a. 1995
WOO, Steven C.; OHARA, Moriyoshi; TORRIE, Evan; SINGH, Jaswinder P.; GUPTA, Anoop: The SPLASH-2 Programs: Characterization and Methodological Considerations. In: *Proceedings of the 22nd International Symposium on Computer Architecture*. Santa Margherita Ligure, Italy, 1995, S. 24–36

Wright 2002
WRIGHT, William: *General linear methods with inherent Runge-Kutta stability*, University of Auckland, Diss., 2002

Zonneveld 1963
ZONNEVELD, J. A.: Automatic integration of ordinary differential equations / Mathematisch Centrum. Postbus 4079, 1009AB Amsterdam, 1963 (R743). – Forschungsbericht. 1964 als Buch veröffentlicht.

Zonneveld 1964
ZONNEVELD, J. A.: *Automatic numerical integration*. Amsterdam, The Netherlands, CWI, Diss., 1964. – Math. Centre Tracts 8

Stichwortverzeichnis

C

D

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z