Effizientes Lösen von Anfangswertproblemen gewöhnlicher Differentialgleichungssysteme mithilfe von Autotuning-Techniken

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

von

Natalia Kalinnik

aus Kiew

1. Gutachter: Prof. Dr. Thomas Rauber
2. Gutachter: Prof. Dr. Claudia Fohry

Tag der Einreichung: 24.02.2015
Tag des Kolloquiums: 23.04.2015

Danksagung

An dieser Stelle möchte ich mich bei allen, die zum Gelingen dieser Arbeit beigetragen haben, bedanken.

In erster Linie gilt mein Dank Herrn Prof. Dr. Thomas Rauber für die Möglichkeit der Promotion in diesem interessanten Forschungsprojekt, für das entgegengebrachte Vertrauen und seine stete Unterstützung. Mein besonderer Dank richtet sich auch an meinen Kollegen PD. Dr. Matthias Korch für die fruchtbaren Diskussionen und die daraus entstandenen Anregungen sowie ein offenes Ohr bei Problemen und Fragen. Ich danke all meinen Kollegen am Lehrstuhl für Angewandte Informatik 2 für das angenehme und freundschaftliche Arbeitsklima. Ein großes Dankeschön gilt Burkhard Zimmermann für das aufopferungsvolle Korrekturlesen und das Beseitigen grammatikalischer Fehler. Schließlich möchte ich meinem Mann dafür danken, dass er auch in schweren Zeiten für mich da ist.

Abstract

During the last decades, parallel computer systems with tremendous computational power, thousands of processors and deep and complex memory hierarchies have been developed. Even though modern parallel architectures provide great opportunities to solve computationally intensive problems, their complexity confronts software developers with immense challenges, since the performance of programs strongly depends on the characteristics of the target platform, such as multi-core processor design and cache architecture. To obtain optimal performance, applications would have to be tuned for each specific target architecture. But the rapidly growing variety of parallel platforms and the fast time to market makes such manual tuning a time-consuming and costly process. The problem of manual tuning is also compounded by the fact that the performance of many programs depends on input data. Consequently, such programs have to be tuned for every possible set of input data.

Auto-tuning is a promising way to avoid manual tuning. The key idea of auto-tuning is the generation of several implementation variants of an algorithm based on program transformations and optimization techniques such as loop interchange, loop tiling, and scheduling. Then, the variant with the best performance on the target machine is selected from the set of generated implementation variants.

This work targets auto-tuning for the efficient solution of initial value problems (IVPs) of systems of ordinary differential equations (ODEs) on modern computer systems. As an example for solution methods for IVPs, a class of explicit predictor–corrector methods of Runge–Kutta type is considered, which possesses a deeply nested loop structure with potential for different loop transformations and different types of parallelism.

In this work online auto-tuning algorithms for the sequential and the parallel execution of a given method are presented. Online auto-tuning is supported by offline benchmarks and exploits the time-stepping nature of ODE methods to selects suitable parameters for variants and the best implementation variant from a candidate pool at runtime during the first time steps.

The auto-tuning algorithms include the selection of suitable tile sizes for implementation variants containing tiled loops. Suitable tile sizes are selected by a combination of an analytical model, based on the working spaces of the loops and regarding the cache organization of the target platform, and an empirical search.

Kurzfassung

Während der letzten Jahrzehnte sind parallele Computersysteme mit enormer Rechenleistung, Tausenden von Prozessoren und immer tieferen und komplexeren Speicherhierarchien entwickelt worden. Wenngleich ihre Nutzung die Möglichkeit schafft, extrem rechenintensive Probleme zu lösen, stellt ihre zunehmende Komplexität die Software-Entwickler vor immense Herausforderungen, da die Leistung eines Programms sehr stark von den Eigenschaften der Zielplattform, wie Multicore- und Cache-Architektur, abhängt. Aufgrund dessen muss der Programmcode für jede einzelne Zielplattform optimiert werden, damit eine hohe Programmleistung erreicht werden kann.

Die rapid wachsende Vielfalt an unterschiedlichen parallelen Plattformen und ihre schnelle Markteinführung macht jedoch das manuelle Optimieren des Programmcodes, auch manuelles Tuning genannt, zu einem zeitraubenden und kostspieligen Prozess. Zusätzlich wird das manuelle Tuning dadurch erschwert, dass die Leistung vieler Programme von den Eingabedaten abhängt. Folglich müssen solche Programme obendrein für jede mögliche Kombination von Eingabedaten optimiert werden.

Automatisches Tuning von Software (Autotuning) ist eine vielversprechende Technik, um ein manuelles Tuning zu vermeiden. Die Grundidee des Autotunings besteht darin, mehrere Varianten eines Programms basierend auf Programmtransformationen und Optimierungstechniken wie Loop-Interchange, Loop-Tiling oder Scheduling (Ablaufplanung) zu erzeugen. Dann wird aus generierten Varianten die Variante mit der besten Laufzeit auf der Zielplattform gewählt.

Diese Arbeit befasst sich mit dem Entwurf von Autotuning-Techniken zur Beschleunigung der Lösung von Anfangswertproblemen gewöhnlicher Differentialgleichungssysteme auf modernen Computersystemen. Dazu wird eine ausgewählte Klasse von Lösungsverfahren für nicht steife Anfangswertprobleme gewöhnlicher Differentialgleichungen (ODEs), nämlich eine Klasse expliziter Prädiktor-Korrektor-Verfahren vom Runge-Kutta-Typ, betrachtet. Die Umsetzung der Berechnungsvorschrift dieser Verfahren führt zu einer tief verschachtelten Schleifenstruktur, die ein großes Potenzial unterschiedlicher Schleifentransformationen und verschiedener Arten von Parallelität besitzt.

In dieser Dissertation werden Online-Autotuning-Algorithmen für die sequentielle und die parallele Ausführung der betrachteten Verfahren vorgestellt. Das Online-Autotuning wird durch Offline-Benchmarks unterstützt und nutzt die zeitschrittorientierte Berechnungsstruktur von ODE-Verfahren aus, um zur Laufzeit geeignete Programmparameter und die schnellste Implementierungsvariante auf der Zielplattform zu wählen.

Die präsentierten Autotuning-Algorithmen beinhalten eine Methode zur automatischen Auswahl geeigneter Blockgrößen für Implementierungsvarianten mit Loop-Tiling. Geeignete Blockgrößen werden durch eine Kombination aus einem analytischen Modell, basierend auf der Berechnung der Größe von Arbeitsräumen und unter Berücksichtigung von Eigenschaften der Cache-Hierarchie der Zielplattform, und einer empirischen Suche bestimmt.

Inhaltsverzeichnis

Abbildungsverzeichnis **xiii**

Tabellenverzeichnis **xvii**

1. Einleitung **1**
1.1. Zielsetzung der Arbeit 3
1.2. Aufbau der Arbeit 4
1.3. Zugehörige Publikationen 6

Publikationsbeiträge: **6**

2. Grundlagen zur Theorie von gewöhnlichen Differentialgleichungen **7**
2.1. Korrekt gestellte AWP 8

3. Verfahren zur numerischen Lösung gewöhnlicher Differentialgleichungen **10**
3.1. Einschrittverfahren 11
3.2. Mehrschrittverfahren 19
3.3. Prädiktor-Korrektor-Verfahren 20
3.4. Stabilität und steife Differentialgleichungen 20
3.5. Parallele Verfahren zur numerischen Lösung von ODEs 22
3.5.1. Parallele explizite RK-Verfahren 23
3.5.2. Parallele implizite RK-Verfahren 26

4. Automatisches Tuning von Software **27**
4.1. Aktuelle Entwicklungen im Bereich der Rechnerarchitektur 27
4.2. Herausforderungen in der Softwareentwicklung 30
4.3. Das automatische Tuning von Software 31
4.3.1. Offline- versus Online-Autotuning 32
4.3.2. Strategien zur Traversierung des Suchraumes 32
4.3.3. Herausforderungen und Probleme bei der Entwicklung eines Autotuning-Ansatzes 34

5. Darstellung verwandter Arbeiten und die Einordnung der eigenen Arbeit **36**
5.1. Darstellung verwandter Arbeiten 36
5.2. Darstellung der eigenen Arbeit 42
5.3. Wissenschaftliche Einordnung der Arbeit 46

6. Codeoptimierung **47**

6.1. Funktionsweise von Caches und das Prinzip der Lokalität 47
6.2. Techniken zur Steigerung der Lokalität von Programmen 50
6.2.1. Ändern des Datenlayouts zur Optimierung der Lokalität 50
6.2.2. Ändern des Speicherzugriffsverhaltens zur Optimierung der Lokalität 51
6.3. Vektorisierung 55
6.4. Einschränkung der Codeoptimierung mit Hilfe eines Compilers 56

7. Die Anwendbarkeit des Autotuning-Verfahrens 57
7.1. Berechnungsstruktur expliziter PC-Verfahren vom RK-Typ 57
7.2. Kandidatenpool sequentieller Implementierungsvarianten 58
7.2.1. Unterschiedliche Möglichkeiten der Realisierung einer Zeitschrittfunktion 59
7.2.2. Allgemeine Implementierungsvarianten 59
7.2.3. Die spezialisierten Implementierungsvarianten 67
Überlappung der Argumentvektoren: Implementierungen *PipeDb1m* und *PipeDb1mt* 70
Pipelining-Implementierungen: *ppDb1m* und *ppDb1mt* 71
7.3. Laufzeitvergleich der Implementierungsvarianten aus dem Kandidatenpool 74
7.4. Vorüberlegungen zur Entwicklung eines Autotuning-Ansatzes 77
7.5. Der Autotuning-Grundalgorithmus 79
7.6. Erfassen von Laufzeiten einzelner Implementierungsvarianten 80
7.7. Experimentelle Evaluierung des Autotuning-Grundalgorithmus 82
7.8. Zusammenfassung 88

8. Automatische Auswahl von Blockgrößen 89
8.1. Das Problem der optimalen Blockgrößenauswahl 90
8.2. Einfluss der Blockgröße auf die Laufzeit einer sequentiellen Implementierungsvariante 92
8.3. Zielsetzung 93
8.4. Der selbstadaptive ODE-Solver mit integrierter Blockgrößenauswahl . . . 95
8.4.1. Die Zeitschritt-Phasen des selbstadaptiven ODE-Solvers 95
8.4.2. Modellbasierte Blockgrößenvorauswahlphase 96
Anforderungen an das Blockgrößenauswahlmodell 98
Das Blockgrößenvorauswahlmodell 98
8.4.3. Realisierung der Autotuningphase 104
8.5. Experimentelle Evaluierung 105
8.5.1. Experimentelle Validierung modellbasierter Blockgrößenauswahl . 105
Einfluss der Wahl einer Blockgröße auf die Laufzeit von Implementierungsvarianten 105
Bewertung der Qualität der modell-gewählten Blockgrößen 107
Der Zusammenhang zwischen der Anzahl von Cache-Fehlzugriffen und effizienten Blockgrößen 111
8.5.2. Vergleich der Laufzeit nicht-adaptiver Implementierungsvarianten und des selbstadaptiven Solvers 112

8.6. Zusammenfassung 116

9. Ein selbstadaptiver paralleler ODE-Solver für gemeinsamen Adressraum 118

9.1. Programmierung mit gemeinsamen Variablen auf ccNUMA-Architektur . 119
9.2. Kandidatenpool paralleler Implementierungsvarianten 120
9.2.1. Allgemeine Implementierungsvarianten 121
9.2.2. Spezialisierte Implementierungsvarianten 127
9.3. Ein selbstadaptiver paralleler ODE-Solver für gemeinsamen Adressraum . 130
9.3.1. Änderungen im Vergleich zum Autotuner für sequentielle Varianten 131
9.3.2. Die Zeitschritt-Phasen des ODE-Solvers 132
9.3.3. Offline-Profiling-Phase 133
9.3.4. Vorauswahlphase 135
9.3.5. Online-Profiling-Phase 135
9.3.6. Autotuningphase 136
9.3.7. Berechnungsphase 138
9.4. Die Strategie zur Blockgrößenauswahl 138
9.4.1. Einfluss der Blockgröße auf die Performance paralleler Implementierungsvarianten mit Loop-Tiling 138
9.4.2. Blockgrößenauswahl 139
Das Arbeitsraum-Modell zur Vorauswahl von Blockgrößen 139
Empirische Suche auf der Menge vorausgewählter Blockgrößen . . 147
Ermittlung der besten Blockgröße anhand der Mehrheitsentscheidung 150
9.5. Experimentelle Evaluierung 151
9.5.1. Experimentelle Validierung der Blockgrößenauswahlstrategie . . . 151
Detaillierte Untersuchung anhand ausgewählter Testfälle 151
Verifikation der Blockgrößenauswahlstrategie anhand weiterer Testfälle 153
Einfluss der Vektorisierung auf die Wahl geeigneter Blockgrößen . 153
Zusammenfassung der Ergebnisse 156
9.5.2. Vergleich paralleler Implementierungsvarianten im Kandidatenpool 158
9.5.3. Performance des selbstadaptiven Solvers 160
9.6. Zusammenfassung 165

10. Weitere durchgeführte und laufende Arbeiten 167

10.1. Automatische Generierung von Implementierungsvarianten 167
10.2. Automatische Bestimmung der Anzahl von Referenzen 168
10.3. Ein selbstadaptiver paralleler ODE-Solver für verteilten Adressraum . . . 168

11. Zusammenfassung und Ausblick 170

11.1. Zusammenfassung 170
11.2. Ausblick 172

A. Verwendete Testprobleme 177

A.1. Reaktions-Diffusion-Modell Brusselator (BRUSS2D) 177
A.2. Schwingende Saite (STRING) 178
A.3. Medizinisches Akzo-Nobel-Problem (MEDAKZO) 179
A.4. Spitzen-Katastrophe (CUSP) 180
A.5. N-Körper-Standardproblem für Sternhaufen (STARS) 181

B. Verwendete Rechnersysteme 182

B.1. Cluster: 32 Knoten, jeder besteht aus 2 AMD Opteron DP 246 Prozessoren 182
B.2. Hydra: 4 Quad-Core Intel Xeon E7330 Prozessoren 182
B.3. Hydrus: 4 Quad-Core AMD Opteron 8350 Prozessoren 183
B.4. Leo: 4 Zwölfkern AMD Opteron 6172 Prozessoren 183
B.5. 2-Socket-Nehalem-Server: 2 Quad-Core Intel Xeon E5530 Prozessoren . . 183
B.6. SGI UltraViolet: 104 Intel Westmere-EX-Prozessoren 184

Literaturverzeichnis 185

Abbildungsverzeichnis

3.1. Das Richtungsfeld und mögliche Lösungsfunktionen der DGL $y'(t) = 2t$. . . 12
3.2. Das Produktionsgraph zum Butcher-Tableau (3.25) [HNW09a]. 23

6.1. Loop-Interchange (Vertauschen der j-Schleife mit der i-Schleife). 52
6.2. Loop-Tiling der $i-$ und $j-$ Schleifen. 53
6.3. Vollständiges Loop Unrolling einer Schleife. 54
6.4. Teilweises Loop-Unrolling mit Aufrollfaktor 2. 54

7.1. Pseudocode einer Zeitschrittfunktion expliziter PC-Verfahren vom RK-Typ. 58
7.2. Pseudocode eines Zeitschrittes der Implementierungsvariante *A*. 62
7.3. Pseudocode eines Zeitschrittes der Implementierungsvariante *D*. 64
7.4. Loop-Tiling der l-Schleife bei der Implementierungsvariante *Dblock*. . . . 65
7.5. Das Zugriffsmuster der Funktion **f** bei einer beschränkten Zugriffsdistanz [Kor12]. 69
7.6. Illustration der Abspeicherung von Ergebnissen der Funktionsauswertungen bei der Überlappung der Argumentvektoren für $m = 4$, $s = 3$ und $n_B = 10$ [KR07a]. 71
7.7. Berechnungsreihenfolge der Blöcke bei den Pipelining-Implementierungen *ppDb1m* und *ppDb1mt*. Illustration des Pipelinings für $m = 4$ Korrektorschritte. Die Zahlen in den Blöcken geben die Reihenfolge der Berechnungen wieder. Die grauen Blöcke werden in der Initialisierungsphase, die roten in der Sweep-Phase und die gelben Blöcke in der Finalisierungsphase der Pipeline berechnet. 72
7.8. Laufzeit der sequentiellen Implementierungsvarianten pro Zeitschritt und Komponente in Abhängigkeit von der Systemgröße für das BRUSS2D-Testproblem und das Radau IA (5)-Verfahren. 75
7.9. Laufzeit der sequentiellen Implementierungsvarianten pro Zeitschritt und Komponente in Abhängigkeit von der Systemgröße und der Stufenzahl des verwendeten ODE-Verfahrens für das CUSP-Testproblem auf dem AMD Opteron 246 Rechnersystem. (a) Radau IA (5)-Verfahren, (b) Lobatto IIIC (8)-Verfahren. 76
7.10. Grundalgorithmus des Online-Autotunings. 80
7.11. Vergleich der normierten Laufzeiten nicht-adaptiver Implementierungsvarianten und der Autotuning-Implementierung für das BRUSS2D Testproblem und das Radau IA (5)-Verfahren. Die Laufzeiten sind in Abhängigkeit der Systemgröße n aufgetragen. (a) Intel Xeon E7330, (b) AMD Opteron DP 246. 83

7.12. Laufzeit der Implementierungen für ODE-Probleme mit beschränkter Zugriffsdistanz auf dem AMD Opteron DP 246 und dem Intel Xeon E7330 Rechnern. . . . 86

7.13. Laufzeit der Implementierungen für ODE-Probleme mit unbeschränkter Zugriffsdistanz auf AMD Opteron DP 246. . . . 87

8.1. Die normierte Laufzeit der *PipeDb2mt*-Implementierung in Abhängigkeit von der System- und Blockgröße für das BRUSS2D-Problem und das Radau IA (5)-Verfahren: (a) AMD Opteron 246, (b) AMD Opteron 8350. . . . 93

8.2. Prozentualer Laufzeitunterschied zu der schnellsten Implementierungsvariante E für n aus dem Bereich $[1.0 \cdot 10^6, 2.5 \cdot 10^6]$, das BRUSS2D-Problem und das Radau IA (5)-Verfahren. . . . 95

8.3. Der selbstadaptive Zeitschrittalgorithmus mit integrierter Blockgrößenauswahl. . . . 97

8.4. Modellbasierte Vorauswahl von Blockgrößen für allgemeine Implementierungsvarianten mit Loop-Tiling. . . . 103

8.5. Laufzeit der Implementierungsvariante *PipeDb2mt* in Abhängigkeit von der Blockgröße für das Testproblem BRUSS2D mit $n = 4.5 \cdot 10^6$ und dem Lobatto IIIC (8)-Verfahren auf (a) dem AMD Opteron 246 und (b) dem AMD Opteron 8350. . . . 106

8.6. Qualität der modell-gewählten Blockgrößen für die Implementierungsvariante *PipeDb2mt* und das Testproblem BRUSS2D. (a) Opteron DP 246, (b) Opteron 8350, (c) Core 2 Duo E8400. . . . 109

8.7. Qualität der modell-gewählten Blockgrößen ts_{min} und ts_{small} für das Testproblem BRUSS2D mit $n = 4.5 \cdot 10^6$. Rechnersystem: AMD Opteron DP 6172. (a) *PipeDb1mt*, Lobatto IIIC (8), (b) *ppDb1mt*, Lobatto IIIC (8), (c) *PipeDb1mt*, Radau IA (5), (d) *ppDb1mt*, Radau IA (5). . . . 110

8.8. Qualität der modell-gewählten Blockgrößen ts_{min} und ts_{small}. Testproblem: BRUSS2D mit $n = 4.5 \cdot 10^6$. Rechnersystem: AMD Opteron 8350. (a) *PipeDb1mt*, Lobatto IIIC (8), (b) *ppDb1mt*, Lobatto IIIC (8), (c) *PipeDb1mt*, Radau IA (5), (d) *ppDb1mt*, Radau IA (5). . . . 111

8.9. Die normierte Anzahl der L1- und L2-Cache-Fehlzugriffe für unterschiedliche System- und Blockgrößen für die Implementierung *PipeDb2mt* und das Testproblem BRUSS2D auf dem AMD Opteron DP 246 System. . . . 113

8.10. Vergleich der normierten Laufzeiten in Abhängigkeit von der Systemgröße für das Testproblem BRUSS2D und das Lobatto IIIC (8)-Verfahren. (a) AMD Opteron DP 246, (b) AMD Opteron 8350, (c) Intel Core 2 Duo E8400. . . . 114

9.1. Pseudocode eines Zeitschrittes der parallelen Implementierungsvariante *PipeDb2mt* für gemeinsamen Adressraum. . . . 125

9.2. Einfluss verschiedener Barrier-Implementierungen auf die Performance der allgemeinen Implementierungsvariante *A*. In dieser Abbildung werden verglichen: die von der Pthread-Bibliothek bereitgestellte (`pthread_barrier_wait()`) Barrier, eine auf Bedingungsvariablen aufbauende Barrier und eine warteschlangen-basierte, auf aktivem Warten beruhende Barrier [CC05]. Testplattform: SGI UV. Testproblem: BRUSS2D mit $n = 8.0 \cdot 10^6$. Korrektorverfahren: Lobatto III C(8). 126
9.3. Ein optimiertes Kommunikationsmuster für ODE-Systeme mit einer beschränkten Zugriffsdistanz. 127
9.4. Einfluss unterschiedlicher Lockmechanismen auf die Laufzeit der Implementierungsvariante *PipeDb1mt*. In der Abbildung werden Pthread-Mutex (passiv-wartend), Pthread-Spinlock (aktiv-wartend) und ein Spinlock auf Basis atomarer Operationen (aktiv-wartend) miteinander verglichen. Testplattform: SGI UV. Testproblem: BRUSS2D mit $n = 0.5 \cdot 10^6$. Korrektorverfahren: Lobatto III C(8). 128
9.5. Die Arbeitsweise paralleler Pipelining-Varianten. Die Kästchen in der Abbildung entsprechen den Blöcken der Matrizen $Y^{(k)}$ und den Vektoren $\Delta\boldsymbol{\eta}^{(m)}$ und $\Delta\boldsymbol{\eta}^{(m-1)}$. Die Nummerierung gibt die Reihenfolge der Berechnungen wieder. Die fett umrandeten Blöcke werden von den Nachbarthreads benötigt. 129
9.6. Die Zeitschritt-Struktur des selbstadaptiven ODE-Solvers. 134
9.7. Einfluss der Blockgröße auf die Performance von Implementierungsvarianten sowie die Qualität ausgewählter Blockgrößen. Testproblem BRUSS2D mit $n = 8.0 \cdot 10^6$ auf dem AMD Opteron 6172 System und 48 Threads. (a) *PipeDb2mt*, (b) *PipeDb1mt*, (c) *ppDb1mt*. 140
9.8. Einfluss der Blockgröße auf die Performance von Implementierungsvarianten sowie die Qualität ausgewählter Blockgrößen. Testproblem STRING mit $n = 8.0 \cdot 10^6$ auf dem AMD Opteron 6172 System und 48 Threads. (a) *PipeDb2mt*, (b) *PipeDb1mt*, (c) *ppDb1mt*. 141
9.9. Einfluss der Blockgröße auf die Performance von Implementierungsvarianten sowie die Qualität ausgewählter Blockgrößen. Testproblem BRUSS2D mit $n = 8.0 \cdot 10^6$ auf dem SGI UV System und 80 Threads. (a) *PipeDb2mt*, (b) *PipeDb1mt*, (c) *ppDb1mt*. 142
9.10. Einfluss der Blockgröße auf die Performance von Implementierungsvarianten sowie die Qualität ausgewählter Blockgrößen. Testproblem STRING mit $n = 8.0 \cdot 10^6$ auf dem SGI UV System und 80 Threads. (a) *PipeDb2mt*, (b) *PipeDb1mt*, (c) *ppDb1mt*. 143
9.11. Die Hilfsfunktion pre-select_tile_sizes() als Teil der Autotuningphase. Diese liefert anhand des Arbeitsraum-Modells zu jeder Implementierungsvariante *c* eine Menge vorselektierter Blockgrößen. 148
9.12. Hilfsfunktion evaluate_implementation() als Teil der Autotuningphase. Diese nutzt zur Bestimmung einer effizienten Blockgröße für eine Implementierungsvariante *c* eine modellgeführte Suche. 149

9.13. Hilfsfunktion eval_tile_size() als Teil der Autotuningphase. Diese misst die Laufzeit einer Implementierungsvariante c unter Verwendung der Blockgröße B. 150
9.14. Einfluss der Vektorisierung auf die Laufzeit von Blockgrößen und die gewählte Blockgröße. Testproblem: BRUSS2D. Korrektorverfahren: Lobatto III C(8). Testplattform: Intel Core i7-4770 (Haswell), 4 Threads. Compiler: icpc (ICC) 14.0.1. (a) $n = 0.5 \cdot 10^6$, mit Vektorisierung, (b) $n = 8.0 \cdot 10^6$, mit Vektorisierung, (c) $n = 0.5 \cdot 10^6$, ohne Vektorisierung, (d) $n = 8.0 \cdot 10^6$, ohne Vektorisierung. 157
9.15. Speedups der im Kandidatenpool enthaltenen parallelen Implementierungsvarianten für gemeinsamen Adressraum, gemessen auf der Testplattform SGI UV für das STRING-Problem. (a) $n = 0.5 \cdot 10^6$, (b) $n = 8.0 \cdot 10^6$. . . 159

Tabellenverzeichnis

5.1. Übersicht ausgewählter Arbeiten im Autotuning-Bereich. 43

7.1. Allgemeine Implementierungsvarianten. 61
7.2. Spezielle Implementierungsvarianten. 69

8.1. Ausgewählte Arbeitsräume für Implementierungsvarianten mit Loop-Tiling. 101

9.1. Kandidatenpool allgemeiner paralleler Implementierungsvarianten für gemeinsamen Adressraum. 122
9.2. Aktueller Kandidatenpool spezialisierter paralleler Implementierungsvarianten für gemeinsamen Adressraum ($d = d(\mathbf{f})$, $b = \max(d(\mathbf{f}), B)$). 123
9.3. Ausgewählte Arbeitsräume für parallele allgemeine Loop-Tiling-Varianten; dabei ist $d = d(\mathbf{f})$. 145
9.4. Ausgewählte Arbeitsräume für parallele spezialisierte Loop-Tiling-Varianten; dabei ist $d = d(\mathbf{f})$ und $b = \max(B, d(\mathbf{f}))$. 146
9.5. Validierung der Blockgrößenauswahlstrategie auf zwei Testplattformen unter Verwendung des Lobatto III C(8)-Verfahrens. 154
9.6. Validierung der Blockgrößenauswahlstrategie auf dem 2-Socket-Nehalem-System unter Verwendung des Lobatto III C(8)-Verfahrens. 155
9.7. Validierung der Blockgrößenauswahlstrategie auf dem 2-Socket-Nehalem-System unter Verwendung des Radau I A(5)-Verfahrens. 155
9.8. Einfluss der Vektorisierung auf die Laufzeit der besten und der schlechtesten Blockgröße. 157
9.9. Experimentelle Evaluierung des Autotuning-Algorithmus anhand der Testprobleme BRUSS2D und STRING auf dem Leo-System. 162
9.10. Experimentelle Evaluierung des Autotuning-Algorithmus auf der SGI UV für die Testprobleme BRUSS2D und STRING. 163
9.11. Experimentelle Evaluierung des Autotuning-Algorithmus auf dem 2-Socket-Nehalem-System für die Testprobleme BRUSS2D und STRING. 164

B.1. Cache-Parameter des AMD Opteron DP 246 Prozessors. 182
B.2. Cache-Parameter des Intel Xeon E7330 Prozessors. 182
B.3. Cache-Parameter des AMD Opteron 8350 Prozessors. 183
B.4. Cache-Parameter des AMD Opteron 6172 Prozessors. 183
B.5. Cache-Parameter des Intel Xeon E5530 Prozessors. 184
B.6. Cache-Parameter des Intel Westmere-EX-Prozessors. 184

1. Einleitung

Differentialgleichungen sind ein Standardwerkzeug in der mathematischen Modellierung, mit dem sich viele Vorgänge in Physik, Chemie, Elektrotechnik, Biologie, Statistik und anderen Gebieten beschreiben lassen. Man findet Differentialgleichungen oft dort, wo die zeitliche Entwicklung eines Systems modelliert werden soll. Da die wenigsten Differentialgleichungen sich analytisch lösen lassen, werden zur ihrer Lösung meistens numerische Verfahren eingesetzt [GJ09]. Durch die Verfügbarkeit von Hochleistungsrechnern können zunehmend auch große und rechenintensive Modelle in annehmbarer Rechenzeit simuliert, d. h. approximativ durch numerische Integrationsverfahren (als Solver bezeichnet) gelöst werden.

Ein bekanntes Beispiel für rechenintensive Anwendungen sind Klima- und Wettersimulationen. Auch im Bereich der Strömungsmechanik werden numerische Simulationen verwendet, um Modelle zur Beschreibung der Wärmestrahlung, der Thermodiffusion und chemischer Reaktionen mit ausreichender Genauigkeit und Geschwindigkeit lösen zu können [Bre04].

Dass der Bedarf an Rechenleistung zur Durchführung komplexer numerischer Simulationsrechnungen extrem hoch sein kann, wird am Beispiel der Millennium-XXL-Simulation [ASW+13, Jül11] deutlich. Diese bisher größte, jemals durchgeführte, kosmologische N-Körper-Simulation, stellte sogar für Supercomputer eine große Herausforderung dar. Die Simulation berechnet die Entwicklung des Universums von einem Zeitpunkt kurz nach dem Urknall bis heute. Aufgrund der immer noch begrenzten physikalischen Rechenleistung aktueller Supercomputer konnte nicht das komplette Universum simuliert werden, sondern nur ein Ausschnitt davon (ein Raumwürfel mit etwa 10 Milliarden Lichtjahren Kantenlänge und etwa 300 Milliarden Teilchen) [Jül11]. Die Simulation wurde auf der Juropa-Maschine am Supercomputerzentrum Jülich auf 12228 Prozessor-Kernen durchgeführt und benötigte insgesamt etwa 10 Tage, was insgesamt 2,86 Millionen CPU-Stunden entspricht.

Im Zusammenhang mit Differentialgleichungen begegnet man oft Anfangswertproblemen (AWPs) für gewöhnliche Differentialgleichungen oder Differentialgleichungssysteme (engl. ordinary differential equation (ODE)). Beispiele für diese Probleme sind: die Berechnung der Flugbahn einer Rakete beim Wiedereintritt in die Erdatmosphäre [Pla10], die Schwingung einer eingespannten Saite [HNW09a] oder die Beschreibung des Ablaufs einer chemischen Reaktion von zwei Substanzen [LN71]. Zudem entstehen Anfangswertprobleme für Systeme gewöhnlicher Differentialgleichungen durch räumliche Diskretisierung zeitabhängiger partieller Differentialgleichungen [Cas05].

Bei einem Anfangswertproblem für ein System gewöhnlicher Differentialgleichungen soll ausgehend von vorgegebenen Anfangswerten, $\mathbf{y}_0 \in \mathbb{R}^n$ zum Zeitpunkt $t_0 \in \mathbb{R}$, die Lösung $\mathbf{y}(t)$ einer gegebenen Differentialgleichung gefunden werden, für die zusätzlich

$\mathbf{y}(t_0) = \mathbf{y}_0$ gelten soll.

Numerische Verfahren zur Lösung von Anfangswertproblemen gewöhnlicher Differentialgleichungssysteme (ODE-Verfahren) approximieren die Lösungsfunktion $\mathbf{y}(t)$ durch die Berechnung einer Folge von diskreten Zeitschritten $\kappa = 0, 1, 2, \ldots$ [SWP12, GJ09]. Solche Lösungsverfahren werden allgemein als Zeitschrittverfahren bezeichnet. Ausgehend vom Integrationszeitpunkt t_0 und der Anfangsapproximation $\boldsymbol{\eta}_0 = \mathbf{y}(t_0) = \mathbf{y}_0$ wird in jedem Zeitschritt κ eine Näherungslösung $\boldsymbol{\eta}_k$ berechnet.

Die Zeit zur Lösung eines Anfangswertproblems wird außer im Fall einer Echtzeitsimulation nicht vorgegeben, sondern hängt von der Komplexität und der Größe des zu lösenden Differentialgleichungssystems, der vorgegebenen Genauigkeitstoleranz, der Länge des Integrationsintervalls und dem gewählten Integrationsverfahren ab [Glö14]. Daher kann die Lösung eines Anfangswertproblems sehr rechenintensiv sein, d. h. sehr hohen Zeitaufwand und viele Zeitschritte benötigen, was hohe Anforderungen an die Leistungsfähigkeit verwendeter Computersysteme stellt.

Der hohe Rechenbedarf zur Lösung von Anfangswertproblemen erfordert eine effiziente Umsetzung von Integrationsverfahren zur ihrer Lösung auf aktuellen Rechnersystemen. Moderne Prozessoren und Multiprozessorsysteme bieten zwar eine hohe Rechenleistung, ihr Aufbau wird aber immer komplexer. Die Multiprozessorsysteme werden zunehmend heterogen und zeichnen sich durch immer tiefere Speicherhierarchien, hierarchische Verbindungsnetzwerke und die Verfügbarkeit von SIMD-Vektorinstruktionen aus [PH13]. Dadurch bedingt wird es für die Programmentwickler und Wissenschaftler immer schwieriger, ihre Programme so weit zu optimieren oder zu parallelisieren, dass die verfügbare Rechenleistung massiv-paralleler Computer optimal ausgenutzt werden kann.

Die Performance eines (parallelen) Programms hängt sehr stark von den Eigenschaften der Zielplattform ab, wie z. B. der Multicore-Architektur, der Cachearchitektur sowie der Topologie und Bandbreite des Speichers. Aus diesem Grund muss der Programmcode für die Zielplattform optimiert werden, damit eine hohe Programmleistung erreicht werden kann. Das manuelle Optimieren von Programmcode durch den Entwickler ist sehr aufwändig und durch ständig kürzer werdende Entwicklungszyklen von neuen Prozessorgenerationen und ihre stets steigende Komplexität schlichtweg zu teuer und nicht praktikabel. Außerdem ist ein manuell optimierter Programmcode schwer zu warten. Bei Lösungsverfahren zur Lösung von Differentialgleichungen wird der Prozess des manuellen Optimierens noch dadurch zusätzlich erschwert, dass die Programmperformance von den Eingabedaten abhängt. Dies bedeutet, dass der Programmcode nicht nur für jede neue am Markt erscheinende Prozessorgeneration, sondern auch für jede Probleminstanz erneut optimiert werden muss.

Zwar verfügen die aktuellen Compiler über verschiedene Optimierungstechniken, um die Laufzeit eines Programms zu verbessern, verzichten aber öfters darauf, bestimmte Codestellen zu optimieren, damit die Korrektheit von Programmen nicht gefährdet wird [Aho08].

Das manuelle Optimieren des Programmcodes kann durch das automatische Software-Tuning (Autotuning) vermieden werden. Beim Autotuning, das seit einiger Zeit vor allem im wissenschaftlichen Bereich eingesetzt wird, wird das Programm automatisch auf

das vorgegebene Ziel (z. B. schnelle Ausführungszeit, geringer Energieverbrauch) hin optimiert. Autotuning wurde erfolgreich für einige mathematische Basisalgorithmen, wie Matrixmultiplikation [WPD01, BACD97] und FFT [FJ05] eingesetzt, für ODE-Verfahren aber bisher nicht.

Den Kernpunkt dieser Dissertation bildet die Entwicklung von Autotuning-Algorithmen und Techniken zum effizienten numerischen Lösen von Anfangswertproblemen gewöhnlicher Differentialgleichungssysteme auf modernen parallelen Architekturen. Die entworfenen Algorithmen erlauben eine automatische Anpassung von ODE-Verfahren an spezifische Eigenschaften der Zielplattform und das zu lösende Anfangswertproblem. Die Anwendbarkeit von Autotuning-Techniken auf ODE-Verfahren wird anhand einer ausgewählten Klasse von Lösungsverfahren für nicht steife Anfangswertprobleme gewöhnlicher Differentialgleichungen, nämlich einer Klasse expliziter Prädiktor-Korrektor-Verfahren (PC-Verfahren) vom Runge-Kutta-Typ, gezeigt.

1.1. Zielsetzung der Arbeit

Das Ziel dieser Arbeit ist die Beschleunigung der Lösung von Anfangswertproblemen gewöhnlicher Differentialgleichungssysteme durch Anwendung des automatischen Software-Tunings und Ausnutzung verfügbarer Rechenleistung paralleler Rechnerarchitekturen. Dadurch soll die Rechenzeit zur Lösung von Probleminstanzen verkürzt werden.

Um dieses Ziel zu erreichen, sollen in dieser Dissertation Autotuning-Algorithmen und Techniken zum numerischen Lösen von Anfangswertproblemen gewöhnlicher Differentialgleichungssysteme auf aktuellen Prozessor- und Rechnerarchitekturen entworfen werden. Im Fokus dieser Arbeit stehen Rechner mit gemeinsamem Adressraum, zu welchen sowohl größere ccNUMA-Rechnersysteme, wie z. B. die SGI UV 2000 mit 2048 Prozessorkernen [Cor14], als auch kleinere Multicore-Systeme zählen. Die entwickelten Autotuning-Techniken lassen sich aber auch auf Systeme mit verteiltem Speicher übertragen, was in dieser Dissertation ebenfalls gezeigt wird.

Durch den Einbau von Autotuning-Techniken sollen die Implementierungsvarianten der ODE-Verfahren sich automatisch an die Zielplattform und das zu lösende Anfangswertproblem anpassen können, indem eine effiziente Implementierungsvariante und geeignete Programmparameter automatisch ausgewählt werden.

Die Berechnungsstruktur von ODE-Verfahren erlaubt i. d. R. die Generierung einer Vielzahl von Implementierungsvarianten, die sich zwar in der Reihenfolge verwendeter Schleifen und den Datenstrukturen unterscheiden, jedoch numerisch äquivalent sind. Aufgrund von Unterschieden in der Schleifenstruktur ändert sich die Reihenfolge der Speicherzugriffe, was dazu führt, dass die Implementierungsvarianten ein unterschiedliches Laufzeit- und Skalierbarkeitsverhalten aufweisen. Welche Implementierungsvariante mit welchen Parametern (z. B. Blockgröße für Loop-Tiling) die schnellste ist, hängt von den Eigenschaften der Zielplattform wie der Cachearchitektur und der Charakteristik des zu lösenden Anfangswertproblems ab.

Da der Aufwand für eine manuelle Auswahl von Implementierungsvarianten und Programmparametern aufgrund der Abhängigkeit von vielen Faktoren (Eigenschaften der

Zielplattform und des Testproblems, verwendetes Integrationsverfahren) sehr hoch sein kann, sollen in dieser Arbeit Autotuning-Algorithmen entwickelt werden, die für ein zu lösendes Anfangswertproblem und eine gegebene Zielplattform eine effiziente Implementierungsvariante und geeignete Parameter automatisch auswählen können.

Es existiert eine Vielzahl unterschiedlicher expliziter und impliziter ODE-Lösungsverfahren (s. Kapitel 3). In dieser Arbeit wird eine Klasse expliziter PC-Verfahren vom Runge-Kutta-Typ betrachtet. Die betrachtete Klasse expliziter PC-Verfahren vom Runge-Kutta-Typ ist aufgrund ihres großen Parallelitätspotenzials bezüglich des Systems und der Methode gewählt worden.

Die Entwicklung von Autotuning-Algorithmen soll ausgehend von den bereits existierenden und auf die Lokalität hin optimierten sequentiellen bzw. parallelen Varianten für gemeinsamen Adressraum des betrachteten Lösungsverfahrens erfolgen [KR07b]. Zuerst soll ein Autotuning-Ansatz für sequentielle Varianten des Verfahrens entworfen werden und dessen Anwendbarkeit anhand einer Reihe von Experimenten auf unterschiedlichen Rechnersystemen und für verschiedene Anfangswertprobleme gezeigt werden. Im nächsten Schritt soll ein Autotuning-Ansatz für parallele Implementierungsvarianten für gemeinsamen Adressraum entwickelt werden.

In der Auswahlmenge von Implementierungsvarianten sind unter anderem Varianten mit Loop-Tiling enthalten, deren Laufzeit sehr stark von der gewählten Blockgröße beeinflusst werden kann. Die Optimierung der Blockgröße für Loop-Tiling-Varianten stellt einen wichtigen Beitrag dieser Arbeit dar. In dieser Arbeit soll zuerst untersucht werden, wie stark die Performance von sequentiellen bzw. parallelen Varianten von der gewählten Blockgröße abhängt. Danach soll eine Strategie zur automatischen Blockgrößenbestimmung ausgearbeitet und in den Autotuning-Algorithmus integriert werden. Da der Suchraum möglicher Blockgrößen sehr groß sein kann, muss ein Weg gefunden werden, diesen in möglichst kurzer Zeit durchlaufen zu können.

Der große Suchraum von Programmvarianten stellt allgemein eine Herausforderung für das Implementieren eines Autotuning-Ansatzes dar. Vor allem das Testen sehr langsamer Implementierungsvarianten zur Laufzeit verringert die Performance des resultierenden Autotuning-Algorithmus. In dieser Arbeit soll daher eine Strategie entwickelt werden, mit der die Evaluierung langsamer Implementierungsvarianten zur Laufzeit vermieden wird.

1.2. Aufbau der Arbeit

Die Arbeit ist wie folgt gegliedert:

- Im Kapitel 2 werden mathematische Grundlagen zur Theorie von gewöhnlichen Differentialgleichungen eingeführt. Im darauf folgenden Kapitel 3 werden numerische Verfahren zur ihrer Lösung vorgestellt. Diese Grundlagen tragen zum Verständnis der in dieser Dissertation vorgestellten Konzepte und Algorithmen bei.

- Kapitel 4 stellt die wichtigsten Grundlagen zum automatischen Tuning von Software (Autotuning) dar und motiviert seine Notwendigkeit anhand der aktuellen

Entwicklungen im Bereich der Hardwarearchitektur und der sich daraus ergebende Herausforderungen für die Softwareentwicklung. Es folgt die Definition des Autotuning-Problems und die anschließende Unterteilung der Autotuning-Ansätze nach dem Zeitpunkt der Evaluierung von Implementierungsvarianten und Parametern. Des Weiteren werden die Strategien zur Traversierung des Suchraumes beschrieben und die allgemeinen Herausforderungen und Probleme bei der Entwicklung eines Autotuning-Ansatzes skizziert.

- Der Überblick über verwandte Arbeiten anderer Autoren wird im Kapitel 5 gegeben. Außerdem wird die eigene Arbeit in die aktuelle Forschungslandschaft eingeordnet.

- Kapitel 6 gibt einen Überblick über den Aufbau und die Funktionsweise von Caches und erläutert das Prinzip der zeitlichen und der räumlichen Lokalität von Programmen. Es folgt die Darstellung der Techniken zur Optimierungen der Speicherzugriffslokalität und die Beschreibung der Programmoptimierung mittels Vektorisierung. Anschließend wird auf die Grenzen der Codeoptimierung durch Compiler eingegangen.

- Im Kapitel 7 wird untersucht, ob eine automatische Selektion von sequentiellen Implementierungsvarianten der betrachteten ODE-Verfahren während der Laufzeit möglich ist. Darauf aufbauend wird der entwickelte Autotuning-Grundalgorithmus für die sequentiellen Implementierungsvarianten vorgestellt, der zwar eine schnelle Implementierungsvariante auswählen kann, aber noch keine Programmparameter optimiert. Die Anwendbarkeit des Algorithmus wird anhand von Laufzeitexperimenten auf unterschiedlichen Zielarchitekturen und für verschiedene Testprobleme gezeigt.

- Im Kapitel 8 wird der Autotuning-Grundalgorithmus um eine automatische Auswahl von Blockgrößen für Loop-Tiling-Varianten erweitert. In anschließenden Experimenten wird die Güte ausgewählter Blockgrößen systematisch auf verschiedenen Rechnersystemen und für unterschiedliche Implementierungsvarianten und Systemgrößen evaluiert.

- Im Kapitel 9 wird zunächst beschrieben, welche Änderungen zur Laufzeitverbesserung an den bereits existierenden parallelen Implementierungsvarianten aus [KR07b] vorgenommen wurden und es werden neue parallele Implementierungsvarianten vorgestellt. Danach wird ein Autotuning-Algorithmus zur automatischen Auswahl paralleler Implementierungsvarianten und zugehöriger Parametern für gemeinsamen Adressraum präsentiert. Die geänderte Strategie der Blockgrößenauswahl wird dargestellt und eine Laufzeitvorhersage zum Ausschluss langsamer Implementierungsvarianten beschrieben.

- Im Kapitel 10 werden weitere durchgeführte und laufende Arbeiten zur Erweiterung des Autotuning-Frameworks präsentiert.

- Kapitel 11 fasst die Dissertation zusammen und gibt einen Ausblick auf mögliche zukünftige Forschungsaufgaben.

1.3. Zugehörige Publikationen

Nachfolgend wird eine Zusammenstellung von Publikationen gegeben, die im Rahmen dieser Dissertation entstanden sind.

Publikationsbeiträge:

[KKR10] Natalia Kalinnik, Matthias Korch und Thomas Rauber: *Applicability of Dynamic Selection of Implementation Variants of Sequential Iterated Runge-Kutta Methods*, in *2010 IEEE International Conference on Cluster Computing – Workshops and Tutorials*, IEEE, 2010.

[KKR11a] N. Kalinnik, M. Korch und T. Rauber: *Dynamic Selection of Implementation Variants of Sequential Iterated Runge-Kutta Methods with Tile Size Sampling*, in *Proceeding of the Second Joint WOSP/SIPEW International Conference on Performance Engineering*, S. 189–200, ACM, 2011.

[KKR11b] N. Kalinnik, M. Korch und T. Rauber: *An efficient time-step-based self-adaptive algorithm for predictor-corrector methods of Runge-Kutta type*, *J. Computational Applied Mathematics*, 236(3):394–410, 2011.

[KKR14] N. Kalinnik, M. Korch und T. Rauber: *Online auto-tuning for the time-step-based parallel solution of ODEs on shared-memory systems*, *J. Parallel Distrib. Computing*, 74(8):2722–2744, 2014.

2. Grundlagen zur Theorie von gewöhnlichen Differentialgleichungen

Differentialgleichungen (DGL) oder **Systeme von Differentialgleichungen** sind ein Standardwerkzeug in der mathematischen Modellierung, mit dem sich viele Vorgänge in der Natur, der Technik, der Physik, der Biologie oder in der Wirtschaft beschreiben lassen. Eine *DGL* ist eine Gleichung, die eine gesuchte Funktion von einer oder mehreren Variablen sowie mindestens eine Ableitung dieser Funktion enthält.

Nachfolgend werden nur einige Bespiele für die Verwendung von DGL in unterschiedlichen Bereichen der Naturwissenschaft und Technik genannt:

- Beschreiben von Bewegungen verschiedener Himmelskörper, Satellitenbahnberechnung,
- Beschreiben von zeitlichen Abläufen chemischer Reaktionen (Reaktionskinetik),
- Simulation des Molekülverhaltens unter vorgegebenen Umgebungsbedingungen wie z. B. der Temperatur (Moleküldynamik),
- Simulation elektrischer Schaltkreise,
- Wachstumsmodelle zum Berechnen einer Population (z. B. Bakterienpopulation),
- Vorhersage des postoperativen Erscheinungsbildes.

Hängt die gesuchte Funktion nur von einer veränderlichen Variablen ab, so spricht man von einer **gewöhnlichen DGL**. Bei vielen physikalischen Vorgängen jedoch hängt die gesuchte Funktion von mehreren Veränderlichen ab. Eine Differentialgleichung, die Ableitungen der gesuchten Funktion nach mehreren Variablen enthält, wird als **partielle DGL (engl. Partial Differential Equation (PDE))** bezeichnet. Die **Ordnung** einer DGL ist gegeben durch die höchste in der DGL vorkommende Ableitung.

Definition 1. *Eine* ***gewöhnliche DGL (engl. Ordinary Differential Equation (ODE)) erster Ordnung*** *im* $\mathbb{R}^n$ *ist durch die Gleichung*

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)) \tag{2.1}$$

gegeben, wobei $n \in \mathbb{N}$ *die Dimension der DGL angibt und* $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ *eine gegebene Funktion ist, auch Vektorfeld genannt.*

Eine Lösung von (2.1) ist eine stetig differenzierbare Funktion $\mathbf{y} : \mathbb{R} \to \mathbb{R}^n$, die (2.1) erfüllt. In der Fachliteratur wird eine DGL der Dimension $n > 1$ auch als **Differentialgleichungssystem** bezeichnet.

Nicht jede DGL ist lösbar, in der Regel kann eine DGL aber eine ganze Kurvenschar von Lösungen besitzen (siehe Beispiel 3.1). In dieser Arbeit werden Anfangswertprobleme für gewöhnliche Differentialgleichungssysteme betrachtet. Große Systeme gewöhnlicher Differentialgleichungen können u. a. durch räumliche Diskretisierung zeitabhängiger PDEs entstehen. Die numerische Lösung von Anfangswertproblemen für zeitabhängige PDEs erfordert zunächst eine Diskretisierung bezüglich Raum und Zeit [DB02]. Durch die räumliche Diskretisierung zeitabhängiger PDEs entsteht ein großes System gewöhnlicher Differentialgleichungen, dieses kann dann mit einem geeigneten numerischen Verfahren für Anfangswertprobleme für ODEs gelöst werden. Zur räumlichen Diskretisierung zeitabhängiger PDEs kann die **Linienmethode (engl. method of lines)** verwendet werden. Bei der Linienmethode erfolgt die Diskretisierung der partiellen Ableitungen mittels finiter Differenzen oder mit Hilfe der Finite Elemente Methode [SWP12]. Diskretisiert man nicht den Raum, sondern die Zeit zuerst, so erhält man durch die **Rothe-Methode Randwertprobleme** für gewöhnliche oder partielle Differentialgleichungen [DB02].

Im Zusammenhang mit gewöhnlichen Differentialgleichungssystemen will man häufig so genannte Anfangswertprobleme lösen. Diese werden auch in dieser Arbeit betrachtet.

Definition 2. *Ein **Anfangswertproblem (AWP)** für eine gewöhnliche DGL (erster Ordnung) besteht darin, zu einem gegebenen $t_0 \in \mathbb{R}$ und $\mathbf{y}_0 \in \mathbb{R}^n$ eine Lösungsfunktion $\mathbf{y}(t)$ mit $t \in [t_0, t_e]$ zu finden, die die Gleichung (2.1) erfüllt und für die darüber hinaus $\mathbf{y}(t_0) = \mathbf{y}_0$ gilt.*

Eine DGL wird als **explizit** bezeichnet, wenn sie sich nach der höchsten Ableitung auflösen lässt, sonst ist eine DGL **implizit**. Jede explizite DGL höher Ordnung kann durch ein Substitutionsverfahren in ein DGL-System erster Ordnung umgeformt werden (vgl. [DR08, s. 380]). Auch jedes System aus n expliziten DGL der Ordnung m kann man in ein DGL-System von $n \cdot m$ expliziten DGL erster Ordnung umwandeln. Deshalb sind die in dieser Arbeit vorgestellten adaptiven Techniken auch für AWP gewöhnlicher DGL höherer Ordnung anwendbar.

Im nächsten Kapitel wird definiert, wann ein AWP **korrekt gestellt** ist. Nur auf korrekt gestellte AWP können numerische Verfahren sinnvoll angewandt werden.

2.1. Korrekt gestellte AWP

Ein AWP heißt **korrekt gestellt (engl. well-posed)**, wenn drei Bedingungen gelten (nach J. Hadamard) [MV06]:

- Das AWP besitzt eine lokale Lösung *(Existenz)*.
- Die lokale Lösung ist eindeutig *(Eindeutigkeit)*.
- Die lokale Lösung hängt stetig von den Anfangswerten ab *(Stabilität)*.

Der Begriff des korrekt gestellten Problems geht auf C. J. Hadamard zurück. Ist eines der oben genannten Bedingungen nicht erfüllt, so spricht man von einem **schlecht gestellten (engl. ill-posed)** Problem.

Für die Existenz einer lokalen Lösung eines AWP reicht die Stetigkeit der rechten Seite $\mathbf{f}(t, \mathbf{y}(t))$ [DR08, Heu06]. Diese Aussage ist in dem Satz von Peano formuliert.

Satz 2.1.1 (**Existenzsatz von Peano** [Heu06]). *Ist die Funktion* $\mathbf{f}$ *in einer Umgebung von* $(t_0, \mathbf{y}_0)$ *stetig, so besitzt das AWP eine Lösung* $\mathbf{y}(t)$ *für* t *nah bei* t_0.

Der Satz von Peano garantiert die *lokale* Existenz einer Lösung in der Nähe des Anfangswertes. Um die Eindeutigkeit der Lösung in der Nähe des Anfangswertes zu garantieren, muss noch eine weitere Bedingung an das AWP gestellt werden.

Der folgende Satz von Picard und Lindelöf sichert die 2. Forderung korrekt gestellter AWP.

Satz 2.1.2 (**Picard und Lindelöf** [Heu06]). *Die Funktion* $\mathbf{f}$ *sei auf dem Gebiet*

$$G := \{(t, \mathbf{y}) :\mid t - t_0 \mid \leq a, ||\mathbf{y} - \mathbf{y}_0|| \leq b\}, \quad a, b > 0. \tag{2.2}$$

stetig. Darüber hinaus ist $\mathbf{f}$ *auf* G *bezüglich* $\mathbf{y}$ ***Lipschitz-stetig***, *d. h. es gibt eine Konstante* $L > 0$ *, so dass*

$$||\mathbf{f}(t, \mathbf{u}) - \mathbf{f}(t, \mathbf{v})|| \leq L \cdot ||(\mathbf{u} - \mathbf{v})|| \quad \textit{für alle} \quad (t, \mathbf{u}), (t, \mathbf{v}) \in G \tag{2.3}$$

gilt. Dann existiert eine eindeutige Lösung $\mathbf{y}$ *von (2.1) in einer Umgebung von* t_0. *In der Definition 2.3 ist* $||\cdot||$ *eine beliebige feste Norm auf* $\mathbb{R}^n$.

Die 3. Forderung der Stabilität bedeutet, dass kleine Änderungen von Anfangsdaten nur geringfügig die Lösung eines AWP beeinflussen sollen. Unter den Voraussetzungen des Satzes von Picard und Lindelöf hängt die Lösung $\mathbf{y}$ stetig von Anfangsdaten $(t_0, \mathbf{y}_0)$ ab.

Satz 2.1.3 (**Stetige Abhängigkeit von den Anfangsdaten**[SWP12]). *Ist* $\mathbf{f}$ *auf dem Gebiet* G *(wie in (2.2) definiert) stetig und erfüllt* $\mathbf{f}$ *auf* G *bezüglich* $\mathbf{y}$ *die Lipschitz-Bedingung (2.3), dann gilt für zwei in* G *verlaufenden Lösungen* $\mathbf{y}(t)$ *und* $\mathbf{z}(t)$ *zu den Anfangswerten*

$$\mathbf{y}(t_0) = \mathbf{y}_0, \, \mathbf{z}(t_0) = \mathbf{z}_0$$

die Abschätzung:

$$||\mathbf{y}(t) - \mathbf{z}(t)|| \leq \exp((L|t - t_0|)) \cdot ||\mathbf{y}_0 - \mathbf{z_0}|| \,. \tag{2.4}$$

Der vorhergehende Satz besagt, dass die Ungenauigkeit in den Anfangsdaten sich höchstens exponentiell fortpflanzen kann (die Abschätzung ergibt sich aus dem Lemma von Gronwall 1918, vgl. [SWP12] s. 7). Für große L und große Unterschiede $|t - t_0|$ kann die Ungenauigkeit der Lösung hingegen ziemlich groß werden.

Es gibt viele Probleme, z. B. aus der Physik (inverse Probleme in der Streuung von Quantenobjekten [CS89]), die die Eigenschaften korrekt gestellter Probleme verletzen [Kab08]. Solche Probleme müssen dann mit Hilfe von Regularisierungsverfahren umformuliert werden [Tik95, Kas06].

3. Verfahren zur numerischen Lösung gewöhnlicher Differentialgleichungen

Im Allgemeinen lässt sich die Lösung eines AWP nur für wenige Probleme mit Hilfe analytischer Verfahren exakt bestimmen. Für die in der Praxis vorkommenden AWP, wie z. B. bei Simulationen von Systemen und Schaltungen, werden deshalb numerische Verfahren verwendet, die das AWP durch eine hinreichend genaue Approximation der Lösungsfunktion lösen. Numerische Verfahren zur Lösung von AWP für ODEs diskretisieren das Integrationsintervall mit Hilfe eines Integrationsgitters

$$t_0 < t_1 < \dots t_N = t_e, \quad I = [t_0, \dots, t_N]$$

und berechnen zu jedem Integrationsknoten (oder Zeitpunkt) t_κ eine Näherungslösung

$$\boldsymbol{\eta}(t_\kappa) \approx \mathbf{y}(t_\kappa), \quad \kappa = 0, \dots, N.$$

Die Näherungslösungen bilden die Funktionswerte einer **Gitterfunktion** $\boldsymbol{\eta}(t) : I \to \mathbb{R}^n$ mit $\boldsymbol{\eta}(t_\kappa) \approx \mathbf{y}(t_\kappa)$. Die Gitterfunktion stellt eine Approximation der gesuchten Lösungsfunktion auf dem gewählten Integrationsintervall dar. Die Anzahl der Gitterpunkte N wird durch die Wahl des Gitters bestimmt. Die Differenz

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

wird als **Schrittweite** bezeichnet. Die zum Zeitpunkt $t_{\kappa+1}$ berechnete Näherung hängt von der Schrittweite h_κ ab. Die Schrittweite h_κ muss dabei nicht unbedingt konstant sein, dies wird aber zugunsten einer einfachen Schreibweise ignoriert und

$$\boldsymbol{\eta}_\kappa = \boldsymbol{\eta}^{(h_{\kappa-1})}(t_\kappa) \tag{3.1}$$

gesetzt. Nachfolgend wird die abgekürzte Schreibweise (3.1) für die Approximation der Lösungsfunktion im Schritt κ zum Zeitpunkt t_κ verwendet. Es existieren zahlreiche numerische Verfahren zur Lösung von gewöhnlichen Differentialgleichungen. Zum Vergleich numerischer Verfahren können folgende Kriterien herangezogen werden [SWP12]:

- Rechenzeit: Als Maß für die Rechenzeit eines numerischen Verfahrens wird oft die Anzahl von Funktionsauswertungen der rechten Seite $\mathbf{f}$ verwendet.
- Genauigkeit: Die Genauigkeit gibt an, wie nah die berechnete Approximation an der exakten Lösung bei einer verwendeten Schrittweite ist. Meistens wird gefordert, dass der lokale Fehler unter einer vom Nutzer vorgegeben Fehlertoleranz liegt (siehe Kapitel 3.1).

- Stabilität: Sie bedeutet, dass der globale Diskretisierungsfehler bei einem stabilen System und einer langen Simulationsdauer begrenzt bleibt. Bei expliziten Verfahren hängt die Stabilität von der Schrittweite und den Eigenwerten des Systems ab [Cra12]. Explizite Verfahren besitzen ein beschränktes Stabilitätsgebiet (siehe Kapitel 3.4).

3.1. Einschrittverfahren

Bei **Mehrschrittverfahren** hängt die Verfahrensvorschrift $\varPhi$ von mehreren zuvor berechneten Approximationen ab. **Einschrittverfahren (ESV)** verwenden zur Berechnung des neuen Approximationswertes $\boldsymbol{\eta}_{\kappa+1}$ nur die zuletzt berechnete Approximation $\boldsymbol{\eta}_{\kappa}$ und sind gegeben durch eine stetige Abbildung

$$\varPhi : \mathbb{R} \times \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n,$$

mit der zu jedem Gitter I und jedem Anfangswert t_0 gemäß folgender Vorschrift

$$\boldsymbol{\eta}_0 = \mathbf{y}_0, \quad \boldsymbol{\eta}_{\kappa+1} = h_\kappa \cdot \varPhi(t_\kappa, \boldsymbol{\eta}_\kappa, h_\kappa) \quad \text{für} \quad \kappa = 0, \ldots, N-1 \tag{3.2}$$

eine Gitterfunktion berechnet wird.

Die Abbildung $\varPhi$ heißt **Verfahrens- oder Inkrement-Vorschrift**. Anhand der Verfahrensvorschrift unterscheidet man zwischen **expliziten** und **impliziten** Integrationsverfahren. Bei einem expliziten Verfahren ist $\varPhi$ von $\boldsymbol{\eta}_{\kappa+1}$ unabhängig und das Gleichungssystem kann durch einfaches Einsetzen gelöst werden. Bei impliziten Verfahren dagegen hängt $\varPhi$ von $\boldsymbol{\eta}_{\kappa+1}$ ab, so dass in jedem Zeitschritt ein nicht-lineares Gleichungssystem gelöst werden muss, um $\boldsymbol{\eta}_{\kappa+1}$ zu bestimmen.

Ein einfaches explizites Einschrittverfahren ist das **Euler-Verfahren** [EES13, Heu06], das im Folgenden näher beschrieben wird.

Euler-Verfahren

Man betrachtet eine explizite DGL erster Ordnung in der Form (2.1). Für die Lösungsfunktion $\mathbf{y}(t)$ auf dem Intervall I gilt:

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)), \forall t \in I \tag{3.3}$$

d. h. die Steigung von $\mathbf{y}(t)$ ist in jedem der Punkte $(t, \mathbf{y}(t))$ durch den jeweiligen Funktionswert $\mathbf{f}(t, \mathbf{y}(t))$ gegeben. Die Steigung $\mathbf{y}'(t)$ gibt der Lösungsfunktion eine Richtung in allen ihren $(t, \mathbf{y}(t))$ Punkten vor. Zeichnet man in jedem Punkt die Richtung der Kurventangente durch einen kleinen Teil dieser Tangente mit der dort vorgegebenen Steigung $\mathbf{y}'(t)$, auch **Linienelement** genannt, so erhält man ein **Richtungsfeld**. Graphisch kann man sich das Lösen von AWP der Form (2.1) so veranschaulichen, dass man nach einer Kurve sucht, die von dem Anfangswert $(t_0, \mathbf{y}_0)$ ausgeht und dem Richtungsfeld der Lösungsfunktion folgt.

Beispiel 3.1. *Die Abbildung 3.1 gibt das Richtungsfeld der Differentialgleichung*

$$y'(t) = 2t$$

an. Die Tangentenrichtung ist in jedem der Punkte $(t, \mathbf{y}(t))$ *durch einen kleinen Pfeil (Linienelement) angedeutet. Die DGL besitzt eine Kurvenschar von Lösungen der Form* $y(t) = t^2 + C$. *In der Abbildung sind nur drei mögliche Lösungsfunktionen (rot dargestellt) für* $C = 0$, $C = 1$, $C = -1$ *eingezeichnet.*

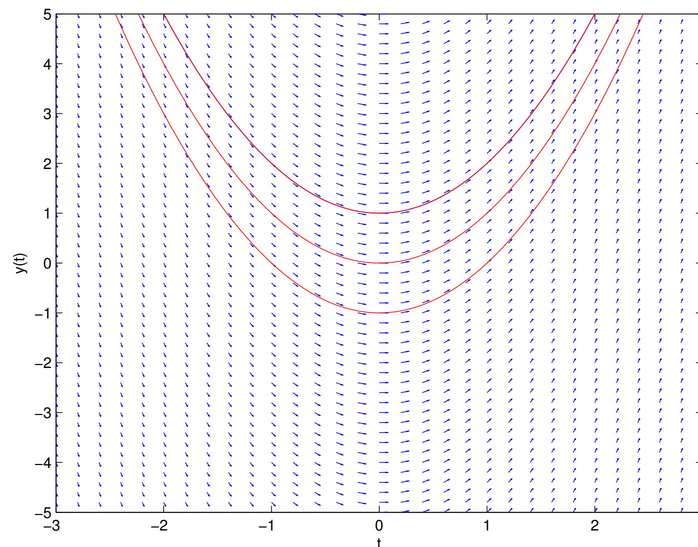


Abbildung 3.1.: Das Richtungsfeld und mögliche Lösungsfunktionen der DGL $y'(t) = 2t$.

Aus der graphischen Beschreibung zum Lösen von AWP gewöhnlicher DGL durch Richtungsfelder lässt sich eine einfache Methode zur näherungsweisen Bestimmung einer Lösung herleiten. Das **Euler-Verfahren** basiert auf folgender Idee: Wir definieren eine Zerlegung auf dem Integrationsintervall $I = [t_0, t_e]$:

$$t_0 < t_1 < \dots t_N = t_e, \quad I_n = [t_{n-1}, t_n]$$

Die Tangentensteigung im Punkt t_κ kann mit Hilfe von Vorwärts-Differenzenquotienten

$$\frac{\boldsymbol{\eta}_{\kappa+1} - \boldsymbol{\eta}_\kappa}{h_\kappa} \approx \mathbf{f}(t_\kappa, \boldsymbol{\eta}_\kappa)$$

approximiert werden. Nach $\boldsymbol{\eta}_{\kappa+1}$ aufgelöst und ausgehend vom Startwert $(t_0, \boldsymbol{\eta}_0)$ erhalten wir die Iterationsvorschrift des Euler-Verfahrens:

$$\boldsymbol{\eta}_{\kappa+1} = \boldsymbol{\eta}_\kappa + h_\kappa \cdot \mathbf{f}(t_\kappa, \boldsymbol{\eta}_\kappa) \tag{3.4}$$

Das bisher betrachtete Euler-Verfahren wurde durch den Vorwärts-Differenzenquotienten realisiert. Hier hängt der Wert $\boldsymbol{\eta}_{\kappa+1}$ nur vom vorher bestimmten Wert $\boldsymbol{\eta}_{\kappa}$ ab, man spricht dabei vom **expliziten Euler-Verfahren**. Approximiert man dagegen die Ableitung in der DGL $\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t))$ durch den Rückwärts-Differenzenquotienten

$$\frac{\boldsymbol{\eta}_{\kappa} - \boldsymbol{\eta}_{\kappa-1}}{h_{\kappa-1}} ,$$

so erhält man das **implizite Euler-Verfahren**:

$$\boldsymbol{\eta}_{\kappa+1} = \boldsymbol{\eta}_{\kappa} + h_{\kappa} \cdot \mathbf{f}(t_{\kappa+1}, \boldsymbol{\eta}_{\kappa+1}). \tag{3.5}$$

Im Gegensatz zum expliziten Euler-Verfahren, hängt der neue Wert $\boldsymbol{\eta}_{\kappa+1}$ nicht nur vom $\boldsymbol{\eta}_{\kappa}$, sondern auch vom $\boldsymbol{\eta}_{\kappa+1}$ ab.

Fehleranalyse

Jedes numerische Verfahren berechnet eine Näherungslösung. In der Praxis möchte man eine Näherungslösung mit einer gewissen Genauigkeit erhalten. Um die Genauigkeit einzuhalten, muss man wissen, wie nah man denn tatsächlich an der exakten Lösung ist. Um das festzustellen, muss der Fehler abgeschätzt werden.

Der Einfachheit halber wird von der konstanten Schrittweise ausgegangen und

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

für alle κ gesetzt.

Der **lokale Diskretisierungsfehler** gibt an, wie stark die in jedem Zeitschritt numerisch berechnete Näherung von der exakten Lösung abweicht.

Definition 3. *Sei $\boldsymbol{\eta}_{\kappa+1}$ eine Approximation, die das Einschrittverfahren nach Ausführung von (3.2) liefert, wenn mit **exakten** Startvektor $\mathbf{y}(t_{\kappa})$ gestartet wird, d. h.:*

$$\boldsymbol{\eta}_{\kappa+1} = \mathbf{y}(t_{\kappa}) + h \cdot \Phi(t_{\kappa}, \mathbf{y}(t_{\kappa}), h).$$

*Dann ist der **lokale Diskretisierungsfehler** an der Stelle $t_{\kappa+1}$, definiert als:*

$$\epsilon(t_{\kappa+1}) = \epsilon(t_{\kappa} + h) = \mathbf{y}(t_{\kappa+1}) - \boldsymbol{\eta}_{\kappa+1} \ ;$$

wobei $\mathbf{y}(t_{\kappa+1})$ die exakte Lösung an dem Gitterpunkt $t_{\kappa+1}$ ist [SWP12].

Den lokalen Diskretisierungsfehler kann man mit Hilfe einer Taylorreihenentwicklung abschätzen.

Definition 4. *Ein Einschrittverfahren heißt **konsistent von der Ordnung p**, falls für den lokalen Diskretisierungsfehler gilt:*

$$\epsilon(t_{\kappa+1}) \leq \mathcal{O}(h^{p+1}).$$

Lokale Diskretisierungsfehler akkumulieren sich in jedem Zeitschritt zu einem globalen Diskretisierungsfehler.

Definition 5. *Der **globale Diskretisierungsfehler** gibt den Fehler zwischen der gesuchten Lösung* $\mathbf{y}(t)$ *und der **tatsächlich berechneten Näherungslösung*** $\boldsymbol{\eta}_\kappa$ *an (im Vergleich zu Definition 3 wurde* $\boldsymbol{\eta}_\kappa$ *nicht ausgehend vom exakten Vektor* $\mathbf{y}(t_{\kappa-1})$ *berechnet), nach einer Folge von Zeitschritten* $\kappa = 1, \ldots, N$*:*

$$e(t_\kappa) = \mathbf{y}(t_\kappa) - \boldsymbol{\eta}_\kappa.$$

Definition 6. *Ein Verfahren heißt **konvergent**, falls:*

$$\lim_{h \to 0} \|e(t_\kappa)\| = 0.$$

Das heißt, eine Näherungslösung konvergiert gegen die exakte Lösung, wenn die Schrittweite h gegen Null strebt.

Definition 7. *Ein Verfahren hat die **Konvergenzordnung*** p*, wenn für* $h \to 0$ *gilt:*

$$e(t_\kappa) = \mathcal{O}(h^p).$$

Da für ein der Lipschitz-Bedingung genügendes Einschrittverfahren die Konsistenzordnung gleich der Konvergenzordnung ist [SWP12], spricht man allgemein von der **Ordnung** eines Einschrittverfahrens. Je größer die Ordnung des Verfahrens, desto schneller wird der Fehler in der Näherungslösung kleiner, wenn die Schrittweite gegen Null geht. Das Euler-Verfahren hat die Ordnung 1. Allgemein gilt, dass bei einem Verfahren der Ordnung p sich der Fehler um den Faktor 2^{-p} verringert [Boc02]. Beim Euler-Verfahren bewirkt die Halbierung der Schrittweite die Halbierung des globalen Diskretisierungsfehlers. Mit der Halbierung der Schrittweite h wächst aber die Anzahl der Gitterpunkte N, da gilt:

$$N \leq (t_e - t_0)/h.$$

Dies bedeutet, dass sich die Anzahl der Gitterpunkte in etwa verdoppelt. Da aber die Anzahl von Funktionsauswertungen $\mathbf{f}$ durch die Anzahl der Gitterpunkte bestimmt wird, verdoppelt sich auch der Rechenaufwand. Deshalb ist besonders für Anwendungen, bei denen die Funktionsauswertungen sehr teuer sind und die eine hohe Genauigkeit bei längeren Zeitintervallen erfordern, der Einsatz von Euler-Verfahren nicht sinnvoll. Für solche Anwendungen sind Verfahren höherer Ordnung besser geeignet. Eine wichtige Klasse von Einschrittverfahren höherer Ordnung stellen **Runge-Kutta (RK)-Verfahren** dar.

Runge-Kutta-Verfahren

Die Idee von Runge-Kutta-Verfahren basiert darauf, keine partiellen Ableitungen, sondern in jedem Zeitschritt an Zwischenwerten (sog. Stufen) z_j Zwischenlösungen zu berechnen [Boc02]. Dabei gibt j die Anzahl der verwendeten Stufen an. Das Verfahren geht auf die Mathematiker C. Runge (1895) und M. W. Kutta (1901) zurück, die das

Verfahren von Heun verallgemeinert haben. Hinsichtlich der ausführlichen Darstellung zur Konstruktion von RK-Verfahren wird der Leser auf [DB02] verwiesen. Werden von einem RK-Verfahren s Stufen verwendet, so spricht man von einem s-stufigen Verfahren. Ein s-stufiges **explizites RK-Verfahren** ist durch folgende Berechnungsvorschrift gegeben [RR00]:

$$\begin{aligned}
&\mathbf{k}_1 := \mathbf{f}(t_\kappa, \boldsymbol{\eta}_\kappa)\\
&\mathbf{k}_2 := \mathbf{f}(t_\kappa + c_2 h, z_2),\ z_2 := \boldsymbol{\eta}_\kappa + h \cdot a_{21} \cdot \mathbf{k}_1\\
&\mathbf{k}_3 := \mathbf{f}(t_\kappa + c_3 h, z_3),\ z_3 := \boldsymbol{\eta}_\kappa + h \cdot (a_{31} \cdot \mathbf{k}_1 + a_{32} \cdot \mathbf{k}_2)\\
&\quad\vdots\\
&\mathbf{k}_s := \mathbf{f}(t_\kappa + c_s h, z_s),\ z_s := \boldsymbol{\eta}_\kappa + \sum_{i=1}^{s-1} h \cdot a_{si} \cdot \mathbf{k}_i
\end{aligned} \tag{3.6}$$

und die Vorschrift, mit der die nächste Approximation berechnet wird, lautet:

$$\boldsymbol{\eta}_{\kappa+1} = \boldsymbol{\eta}_\kappa + h \cdot \sum_{l=1}^{s} b_l \mathbf{k}_l. \tag{3.7}$$

Ein konkretes Runge-Kutta-Verfahren ist durch die Verfahrenskoeffizienten a_{li}, b_i, c_i charakterisiert, die als **Butcher-Tableau** angeordnet werden können, wobei $A = (a_{li}) \in \mathbb{R}^{s,s}$, $\mathbf{b} = (b_l) \in \mathbb{R}^s$ und $\mathbf{c} = (c_l) \in \mathbb{R}^s$:

$$\begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b}^\mathbf{T} \end{array} = \begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ c_2 & a_{21} & a_{22} & \dots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \end{array} \tag{3.8}$$

Bei expliziten RK-Verfahren sind die Koeffizienten a_{li} für $i \geq l$ gleich Null, andernfalls spricht man von **impliziten RK-Verfahren**. Bei expliziten Verfahren hängt die Lösung des Gleichungssystems (3.6) nur von den bereits berechneten $l-1$ Stufenvektoren $\mathbf{k}_1, \dots, \mathbf{k}_{l-1}$ ab. Bei impliziten Verfahren dagegen muss in jedem Zeitschritt ein nicht-lineares Gleichungssystem der Größe sn gelöst werden, wenn n die Anzahl der Unbekannten ist. Im Folgenden werden die expliziten RK-Verfahren näher betrachtet.

In der Literatur [DB02, SWP12] findet man für die expliziten RK-Verfahren die Angabe von so genannten **Butcher-Schranken** [But64]. Diese geben die minimale Anzahl von Stufen an, die erforderlich ist, um ein RK Verfahren einer bestimmten Ordnung p zu konstruieren. Eine höhere Anzahl von Stufen sollte zu einem Verfahren höherer Ordnung führen. Bei expliziten Verfahren lässt sich allerdings für $s \leq 8$ maximal die Ordnung $p = s - 2$ erreichen. Die impliziten RK-Verfahren können hingegen die Ordnung $p = 2s$ erreichen. Implizite RK-Verfahren, die die maximale Ordnung $2s$ erreichen, basieren auf Gauß-Legendre-Formeln und heißen **Gauß-Verfahren**. Weitere häufig ver-

wendete implizite RK-Verfahren sind **Radau-Verfahren** mit der Ordnung $p = 2s - 1$ und **Lobatto-Verfahren** mit der Ordnung $2s - 2$ [SWP12, DB02]. Bei Radau-Verfahren gilt für die Verfahrenskoeffizienten im Butcher-Tableau $c_1 = 0$ oder $c_s = 1$ und für die Lobatto-Verfahren gilt $c_1 = 0$ und $c_s = 1$. Der Vorteil von Radau- und Lobatto-Verfahren gegenüber den Gauß-Verfahren ist der größere Stabilitätsbereich und die geringe Anzahl impliziter Gleichungen. Bei Lobatto-Verfahren z. B. enthält die erste und die letzte Spalte des Butcher-Tableaus nur Nullen. Deshalb können die Stufen k_1 und k_s explizit berechnet werden. In dieser Arbeit werden Radau IIA [Ehl68] und Lobatto IIIC [Chi71] Verfahren verwendet.

Schrittweitensteuerung

In der Praxis ist man an Lösungsverfahren interessiert, die eine vorgegebene Genauigkeit erreichen und mit möglichst wenig Rechenzeit auskommen. Die Rechenzeit eines numerischen Verfahrens hängt von der Zeit pro Integrationsschritt, aber auch von deren Anzahl ab. Die Anzahl der Integrationsschritte lässt sich durch die Wahl einer geeigneten Schrittweite steuern. Da sich die DGL in unterschiedlichen Abszissenbereichen unterschiedlich verhalten kann, kann die Wahl einer konstanten Schrittweite zu unnötigen Zeitschritten führen. Stattdessen sollte die Schrittweite an das Verhalten einer DGL angepasst werden. So kann h klein gewählt werden an den Bereichen, wo sich die Lösung schnell ändert, und größer für die Bereiche, wo sich die Lösung nur langsam ändert. Durch eine solche **automatische Schrittweitensteuerung** kann unter Umständen die Anzahl der Zeitschritte wesentlich reduziert werden. Um eine vorgegebene Genauigkeit zu erreichen, darf die Schrittweite nicht zu groß gewählt werden. Eine zu kleine Schrittweite allerdings kann die Rechenzeit unnötig in die Höhe treiben. Eine zu klein gewählte Schrittweite kann auch dazu führen, dass sich die Rundungsfehler stark aufsummieren, da die Anzahl der Rundungsfehler proportional zur Anzahl der Schritte ansteigt. Um in jedem Zeitschritt eine geeignete Schrittweite festzulegen, muss zunächst der Fehler abgeschätzt werden. Weil die Abschätzung eines globalen Fehlers ohne das Wissen über die Lösung einer DGL schwer ist, basiert die automatische Schrittweitensteuerung auf der Schätzung des lokalen Fehlers [SWP12]. Der Anwender gibt dabei eine Toleranzgrenze TOL für den lokalen Fehler an, die in jedem Zeitschritt eingehalten werden muss. Für den lokalen Fehler ϵ im Schritt $\kappa + 1$, der mit der Schrittweite h_κ ausgeführt worden ist, gilt dann die Forderung:

$$\|\epsilon(t_{\kappa+1})^{(h_\kappa)}\| \leq TOL. \tag{3.9}$$

Es sei $\bar{\epsilon}$ der Schätzer für den lokalen Fehler in einem Integrationsschritt. Nach der Abschätzung des Fehlers in einem Zeitschritt können zwei Fälle auftreten:

- $\bar{\epsilon} \leq TOL$. Hier wird die erforderliche Genauigkeit erreicht. Der Zeitschritt und die berechnete Approximation werden akzeptiert und die neue Schrittweite h_{neu} berechnet.

- $\bar{\epsilon} > TOL$. Die erforderliche Genauigkeit wird nicht eingehalten. Der Zeitschritt und die entsprechende Approximation $\eta_{\kappa+1}$ werden verworfen, es erfolgt die Wie-

derholung des Zeitschrittes mit einer kleineren Schrittweite h_{neu}.

Für die Abschätzung $\bar{\epsilon}$ des lokalen Fehlers gibt es zwei Methoden: die **Richardson-Extrapolation** und die **Einbettungsstrategie** [SWP12].

Richardson-Extrapolation. Es sei ein s-stufiges explizites RK-Verfahren der Konsistenzordnung p gegeben. Die **Richardson-Extrapolation** verwendet zwei Approximationen des gleichen Einschrittverfahrens, die auf der Basis unterschiedlicher Schrittweiten entstanden sind. Ausgehend von einer Approximation η_κ zum Zeitpunkt t_κ werden zwei Näherungslösungen berechnet. Die erste Näherungslösung $\eta_{\kappa+1}^{(h_\kappa)}$ wird durch die Anwendung der Schrittweite h_κ berechnet. Zusätzlich berechnet man ausgehend von η_κ eine zweite Approximation $\eta_{\kappa+1}^{(2\cdot h_\kappa/2)}$ durch zwei aufeinanderfolgende Schritte mit der Schrittweite $h_\kappa/2$.

Es sei $\mathbf{y}(t_\kappa + h_\kappa)$ die exakte Lösung an der Stelle $t_\kappa + h_\kappa$. Für den lokalen Fehler an der Stelle $t_\kappa + h_\kappa$ gilt nach zwei Schritten mit der Schrittweite $h_\kappa/2$ die Abschätzung [SWP12]:

$$\mathbf{y}(t_\kappa + h_\kappa) - \eta_{\kappa+1}^{(2\cdot h_\kappa/2)} = \frac{\eta_{\kappa+1}^{(2\cdot h_\kappa/2)} - \eta_{\kappa+1}^{(h_\kappa)}}{2^p - 1} + O(h^{p+2}) \tag{3.10}$$

Bei einer automatischen Schrittweitensteuerung soll die Schrittweite h_{neu} für den nächsten Schritt so bestimmt werden, dass der in der Gleichung (3.10) abgeschätzter Fehler eine vorgegebene Genauigkeitsanforderung erfüllt.

Nach ([SWP12]) sind häufig verwendete Normen für die Fehlerabschätzung:

$$\bar{\epsilon} = \sqrt{\frac{1}{n} \cdot \sum_{i=1}^{n} \left(\frac{\eta_{\kappa+1,i}^{(h_\kappa)} - \eta_{\kappa+1,i}^{(2\cdot h_\kappa/2)}}{(2^p - 1) sk_i} \right)^2} \text{oder } \bar{\epsilon} = \max_{i=1,\ldots,n} \frac{|\eta_{\kappa+1,i}^{(h_\kappa)} - \eta_{\kappa+1,i}^{(2\cdot h_\kappa/2)}|}{(2^p - 1) sk_i} \tag{3.11}$$

Dabei sind die Komponenten des Skalierungsvektors **sk** in (3.11) gegeben durch:

$$sk_i = atol_i + rtol_i \cdot \max \left(|\eta_{\kappa,i}|, |\eta_{\kappa+1,i}^{(2\cdot h_\kappa/2)}| \right)$$

Die Toleranzen $atol_i$ und $rtol_i$ werden vom Benutzer vorgegeben. Häufig wählt man die absoluten und relativen Toleranzen $atol_i$ bzw. $rtol_i$ für alle Komponenten $i = 1, \ldots, n$ gleich und gibt nur die absoluten und relativen Toleranzen $ATOL$ und $RTOL$ vor. Die neue Schrittweite h_{neu} wird wie folgt berechnet [SWP12]:

$$h_{neu} = \min \left(\alpha_{max}, \max(\alpha_{min}, \alpha(1/\bar{\epsilon})^{1/(p+1)}) \right) \cdot h. \tag{3.12}$$

Bei der Berechnung der neuen Schrittweite in (3.12) gibt α einen Sicherheitsfaktor an und dient zur Vermeidung von häufigen Schrittwiederholungen. Ein typischer Wert für die Wahl des Sicherheitsfaktors ist z. B. $\alpha = 0,9$. Die $\alpha_{min}-$ und $\alpha_{max}-$ Faktoren verhindern starke Schwankungen der Schrittweite. Für α_{max} sind die Werte zwischen 1,5 und 10 und für α_{min} die Werte zwischen 0,1 und 0,5 gebräuchlich [SWP12].

Einbettungsstrategie. Zur Abschätzung wird ein zweites ESV unterschiedlicher Ordnung verwendet, mit dem eine zweite Approximation bei gleicher Schrittweise berechnet wird. Die Idee basiert darauf, aus zwei Einschrittverfahren unterschiedlicher Ordnung ein eingebettetes RK-Verfahren zu konstruieren, bei dem die Koeffizienten der Stufen a_{li} und c_i beider Verfahren übereinstimmen und sich beide Verfahren nur in den Koeffizienten des Gewichtungsvektors b_i bzw. $\hat{b_i}$ unterscheiden. Ein Paar solcher eingebetteter ESV $(\Phi, \hat{\Phi})$ wird durch ein Butcher-Tableau folgender Form dargestellt:

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ c_2 & a_{21} & a_{22} & \dots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \\ \hline & \hat{b_1} & \hat{b_2} & \dots & \hat{b_s} \end{array} \tag{3.13}$$

Für die Fehlerabschätzung $\bar{\epsilon}$ werden zwei Approximationen erstellt:

$$\boldsymbol{\eta}_{\kappa+1} = \boldsymbol{\eta}_\kappa + h_\kappa \cdot \sum_{l=1}^{s} b_l \mathbf{k}_l \tag{3.14}$$

$$\hat{\boldsymbol{\eta}}_{\kappa+1} = \boldsymbol{\eta}_\kappa + h_\kappa \cdot \sum_{l=1}^{s} \hat{b_l} \mathbf{k}_l. \tag{3.15}$$

Die lokale Fehlerabschätzung ergibt sich als Norm der Differenz beider Approximationen:

$$\bar{\epsilon} = \frac{\|\boldsymbol{\eta}_{\kappa+1} - \hat{\boldsymbol{\eta}}_{\kappa+1}\|}{\mathbf{sk}} \tag{3.16}$$

Die Komponenten des Skalierungsvektors **sk** sind dabei gegeben als:

$$sk_i = atol_i + rtol_i \cdot \max(|\eta_{\kappa,i}|, |\eta_{\kappa+1,i}|).$$

Der Vorteil der Fehlereinschätzung mittels eingebetteter RK-Verfahren gegenüber der Richardson-Extrapolation ist die geringere Anzahl von Funktionsauswertungen und daher auch ein geringerer Rechenaufwand. Bei sehr großen Genauigkeitsanforderungen allerdings stellt die Richardson-Extrapolation eine sinnvolle Alternative dar, weil dadurch ein Verfahren höherer Ordnung verwendet werden kann [SWP12]. Beispiele eingebetteter RK-Verfahren sind die Runge-Kutta-Fehlberg [Feh70] und Dormand-Prince-Verfahren [PD81].

PI-Regler. In der Praxis kann es häufiger vorkommen, dass die Schrittweite sehr stark schwankt und es zu vielen Schrittverwerfungen kommen kann. Daher haben Gustafsson, Lundh und Söderlind (1988) vorgeschlagen, für die Schrittweitensteuerung einen so genannten **PI-Regler (engl. proportional-integral controller)** zu verwenden, um starke Schwankungen der Schrittweite zu vermeiden. Wird die Schrittweitensteuerung

als PI-Regler implementiert, so nutzt man nicht nur den Fehlerschätzer vom aktuellen Zeitschritt $\bar{\epsilon}_j$, sondern auch den Fehlerschätzer aus dem vorherigen Zeitschritt $\bar{\epsilon}_{\kappa-1}$ [SWP12]. Die Bestimmung der neuen Schrittweite h_{neu} in der Gleichung (3.12) ändert sich dann wie folgt [SWP12]:

$$h_{neu} = \left(\frac{1}{\bar{\epsilon}_\kappa}\right)^{\alpha/(q^*+1)} \cdot \left(\frac{1}{\bar{\epsilon}_{\kappa-1}}\right)^{\beta/(q^*+1)} \cdot h, \tag{3.17}$$

wobei q^* die Ordnung des Schätzverfahrens ist. Gustafsson hat in [Gus94] als geeignete Koeffizienten $\alpha = 0,7$, $\beta = -0,4$ vorgeschlagen, jedoch hängt deren Wahl vom verwendeten Verfahren und der zu lösenden ODE ab. Mehr Details zur Schrittweitensteuerung mittels PI-Regler findet man in [Soe06].

3.2. Mehrschrittverfahren

Für Einschrittverfahren ist die Konstruktion von Verfahren einer höheren Ordnung nur durch Erhöhung der Stufenanzahl s und somit nur durch die Erhöhung der Anzahl von Funktionsauswertungen $\mathbf{f}$ möglich. Bei Mehrschrittverfahren dagegen ist das Erreichen einer höheren Ordnung mit gleicher Anzahl von Funktionsauswertungen $\mathbf{f}$ möglich. Im Gegensatz zu Einschrittverfahren hängt bei Mehrschrittverfahren die Verfahrensvorschrift von mehreren vorher berechneten Näherungen ab. Da die Änderung der Schrittweite h sich bei Mehrschrittverfahren schwierig gestaltet, wird zur Formulierung von Mehrschrittverfahren von einer konstanten Schrittweite h ausgegangen. Die **Mehrschrittverfahren** oder **k-Schrittverfahren** sind gegeben durch folgende Form [Kor07]:

$$\boldsymbol{\eta}_{\kappa+k} = \Phi(t_{\kappa+k-1}, \boldsymbol{\eta}_\kappa, \boldsymbol{\eta}_{\kappa+1}, \dots \boldsymbol{\eta}_{\kappa+k}, h_\kappa) \quad \text{für} \quad \kappa = 0, \dots, N-k \tag{3.18}$$

Um die Näherungslösungen zu erhalten, werden k Startwerte $\boldsymbol{\eta}_0, \dots, \boldsymbol{\eta}_{k-1}$ benötigt. Diese werden meistens mit einem Einschrittverfahren berechnet. Bei **linearen Mehrschrittverfahren (LM)** hängt die Verfahrensfunktion Φ linear von $\mathbf{f}$ ab. Lineare Mehrschrittverfahren sind durch folgende Berechnungsvorschrift gegeben [SWP12]:

$$\sum_{l=0}^{k} a_l \boldsymbol{\eta}_{\kappa+l} = h \sum_{l=0}^{k} b_l \mathbf{f}(t_{\kappa+l}, \boldsymbol{\eta}_{\kappa+l}), \quad \kappa = 0, \dots, N-k, \tag{3.19}$$

mit Koeffizienten $a_0, \dots, a_{k-1}$ und $b_0, \dots, b_k$, wobei $|a_0| + |b_0| > 0$ gilt. Durch Normierung des Koeffizienten $a_k = 1$, wird die Formel (3.19) zu

$$\boldsymbol{\eta}_{\kappa+k} = -\sum_{l=0}^{k-1} a_l \boldsymbol{\eta}_{\kappa+l} + h \sum_{l=0}^{k} b_l \mathbf{f}(t_{\kappa+l}, \boldsymbol{\eta}_{\kappa+l}), \quad \kappa = 0, \dots, N-k. \tag{3.20}$$

Ist $b_k \neq 0$, so spricht man von einem impliziten LM. Ist $b_k = 0$, so ist das Verfahren explizit und die Näherungen $\boldsymbol{\eta}_{\kappa+k}$ können durch Einsetzen vorher berechneter Näherungen $\boldsymbol{\eta}_{\kappa+l}$ mit $l = 0, \dots, k-1$ in die Formel (3.20) berechnet werden. Durch die Wahl der

Verfahrenskoeffizienten a_l mit $0 \leq l \leq k-1$ und b_l mit $0 \leq l \leq k$ lassen sich aus der Formel 3.20 spezielle Klassen von Mehrschrittverfahren wie **Adams-Bashforth**, **Adams-Moulton** und **Rückwärtsdifferenzenmethoden** ableiten. Für die k-Schrittverfahren maximaler Ordnung gilt [DB02]:

- Es existiert kein k-Schrittverfahren der Ordnung $2k+1$.
- Ein implizites Verfahren der Ordnung $p = 2k$ und ein explizites k-Schrittverfahren der Ordnung $2k-1$ lassen sich konstruieren.

Jedoch sind diese Verfahren maximaler Ordnung für $k > 1$ meistens ungeeignet, da sie numerisch instabil sind [Tod50, DB02].

3.3. Prädiktor-Korrektor-Verfahren

Prädiktor-Korrektor-Verfahren (PC-Verfahren) versuchen die Vorteile der einfachen Berechnungsstruktur expliziter Verfahren mit besseren Stabilitätseigenschaften und höheren Genauigkeit impliziter Verfahren zu verbinden [Kor07]. Bei einem PC-Verfahren wird zunächst mit Hilfe eines expliziten Verfahrens, auch **Prädiktor** genannt, ein Startvektor berechnet, der dann in mehreren Iterationsschritten mit einem impliziten Verfahren, dem sog. **Korrektor** (auch Basisverfahren genannt), verbessert wird. Ein Beispiel für ein Prädiktor-Korrektor-Verfahren ist das Einschrittverfahren von Heun, das eine Kombination des expliziten Euler-Verfahrens und der Trapezregel darstellt [Völ11].

Weitere Beispiele für Prädiktor-Korrektor-Verfahren sind Mehrschrittverfahren wie Adams-Bashforth-Moulton und Rückwärtsdifferenzenmethoden (engl. BDF). Wird bei einem PC-Verfahren ein RK-Verfahren als Basisverfahren benutzt, so spricht man von **Prädiktor-Korrektor-Verfahren vom RK-Typ (PC-Verfahren vom RK-Typ)**.

3.4. Stabilität und steife Differentialgleichungen

In der Praxis gibt es Systeme von ODEs, die auf Grund von Stabilitätsproblemen sehr kleine Schrittweiten erfordern, um eine genaue Lösung zu erhalten. Solche Systeme werden als **steif** bezeichnet. In der Literatur gibt es keine präzise Definition für den Begriff **steife ODE-Systeme**. Jedoch haben steife Differentialgleichungssysteme oft folgende Eigenschaften [SWP12]:

- Ein System von ODEs ist steif, wenn gewisse Komponenten der Lösung sehr viel schneller abklingen als andere.
- Ein System von ODEs ist steif, wenn explizite Verfahren sehr kleine Schrittweiten verwenden, was gleichzeitig bedeutet, dass die Wahl der Schrittweite nicht durch Genauigkeitsanforderungen, sondern durch Stabilitätseigenschaften des Systems bestimmt wird.

Wegen besonderem Abklingverhalten steifer Systeme ist es wichtig, die unterschiedlich abklingenden Komponenten eines Systems mit geeigneten Schrittweiten wiederzugeben. Um die Eigenschaften eines ODE-Verfahrens bzgl. steifer Systeme zu überprüfen, benutzt man einfache Testprobleme, wie z. B. die folgende Testgleichung von Germund Dahlquist [Dah63, Her06]:

$$y'(t) = \lambda y(t), \quad y(0) = 1, \quad \lambda \in \mathbb{C}, \quad Re(\lambda) \leq 0 \tag{3.21}$$

Die Lösung von (3.21) ist die Lösungsfunktion $y(t) = e^{\lambda t}$. Für $Re(\lambda) \leq 0$ ist $y(t)$ für $t \to \infty$ beschränkt. Wendet man die Testgleichung (3.21) auf ein explizites RK-Verfahren an, so erhält man eine Rekursion:

$$\boldsymbol{\eta}_{\kappa+1} = R(z)\boldsymbol{\eta}_{\kappa}, \tag{3.22}$$

mit $z = \lambda h \in \mathbb{C}$. Die Funktion $R(z)$ ist die **Stabilitätsfunktion** des numerischen Verfahrens. Da die Lösung der Testgleichung für $Re(\lambda) \leq 0$ (d. h. für alle $\lambda \in \mathbb{C}^-$) beschränkt ist, muss auch die Lösung des numerischen Problems die gleiche Eigenschaft haben. Dies ist gewährleistet, wenn $|R(z)| \leq 1$ gilt. Man definiert das **Stabilitätsgebiet** S eines Einschrittverfahrens wie folgt [1]:

$$S = \{|R(z)| \leq 1 \,|\, z \in \mathbb{C}\}. \tag{3.23}$$

Definition 8. *Ein numerisches Verfahren heißt **A-stabil**, falls*

$$|R(z) \leq 1| \quad \text{für} \quad Re(z) \leq 0 \tag{3.24}$$

gilt.

Ist ein Verfahren A-stabil, so umfasst sein Stabilitätsgebiet die komplette linke Halbebene der komplexen Zahlen. Ein A-stabiles Verfahren liefert unabhängig von der Schrittweite h für alle $\lambda \in \mathbb{C}^-$ eine monoton fallende Folge von Näherungen. Das bedeutet, dass bei A-stabilen Verfahren das Verfahren für beliebige Werte von h stabil bleibt und so die Schrittweite h nur gemäß Genauigkeitsanforderungen gewählt werden kann.

Für das s-stufige, explizite RK-Verfahren ist die Stabilitätsfunktion $R(z)$ ein Polynom von höchstens Grad s (für einen Beweis siehe [DR08]). Für Polynome gilt aber, dass mit $z \to \infty$ auch $|R(z)|$ gegen unendlich strebt. Deshalb sind explizite RK-Verfahren nicht A-stabil. Weil die Lösung der Funktion $e^{\lambda h}$ für $\lambda < 0$ sehr schnell abklingt, lässt sich dieses schnelle Abklingverhalten mit einem Polynom nur für kleine Werte $|\lambda h|$ wiedergeben. Bei sehr steifen Problemen wird das explizite Verfahren für große Werte von λ sehr kleine Schrittweiten verlangen, damit $|\lambda h|$ klein genug bleibt. Da bei expliziten Verfahren die Stabilitätsfunktion in der linken komplexen Ebene unbeschränkt ist, sind explizite Verfahren nie A-stabil [DR08]. Für implizite RK-Verfahren ist die Stabilitätfunktion eine rationale Funktion, wobei Zähler und Nenner höchstens vom Grad s sind [DR08].

[1] Dies ist die Definition des Stabilitätsgebiets der Einschrittverfahren, für Mehrschrittverfahren wird das Stabilitätsgebiet S anders definiert [DR08].

Rationale Funktionen können das schnelle Abklingverhalten der exponentiellen Lösungsfunktion (3.21) auch für große Werte von λ gut wiedergeben.

Generell gilt, dass man bei der Integration steifer Systeme A-stabile Verfahren verwenden sollte, da diese die Wahl zu kleinen Schrittweiten verhindern. Da aber implizite Verfahren in jedem Zeitschritt ein nicht-lineares GLS lösen müssen, ist die Durchführung eines Zeitschrittes bei impliziten Verfahren teurer als bei expliziten Verfahren. Der Aufwand zum Lösen eines nicht-linearen GLS in einem Zeitschritt kann jedoch durch die Anwendung einer größeren Schrittweite weitgehend kompensiert werden.

3.5. Parallele Verfahren zur numerischen Lösung von ODEs

Parallele Verfahren zur numerischen Lösung von ODEs lassen sich nach Gear [Gea86, Bur95, GX93] in drei Kategorien unterteilen:

- Bei der **Parallelität bezüglich des Systems** werden unterschiedliche Komponenten eines ODE-Systems parallel berechnet. Besonders für große ODE-Systeme lässt sich durch das Verteilen der Daten eine sehr gute Skalierbarkeit erreichen.

- Bei der **Parallelität bezüglich der Methode** ist die Methode selbst parallelisierbar. Bei dieser Art der Parallelisierung nutzt man die spezielle Struktur der Methode, um die unabhängigen Berechnungen innerhalb eines Zeitschrittes auszuführen. Parallelität bezüglich der Methode bietet typischerweise ein eingeschränktes Skalierungspotenzial, da bei diesem Ansatz die Anzahl verwendeter Prozessoren klein bleibt. Ein Beispiel für die Parallelität bezüglich der Methode bei expliziten Verfahren ist die parallele Berechnung von unabhängigen Stufenvektoren bei expliziten Prädiktor-Korrektor-Verfahren vom RK-Typ (siehe Abschnitt 3.5.1). Bei impliziten Lösungsverfahren gibt es Ansätze zur Parallelisierung innerhalb eines Newtonschrittes auf der Ebene der linearen Algebra. Ein Beispiel dafür ist die parallele Berechnung von LU-Zerlegungen und Rücksubstitutionen innerhalb eines Newtonschrittes bei multi-impliziten RK-Verfahren [Ben96].

- Werden mehrere Zeitschritte parallel ausgeführt, so spricht man von **Parallelität bezüglich der Zeit**. Algorithmen, die Parallelität bezüglich Zeit ausnutzen, sind Parareal [LMT01], PITA [FC03], PFASST [EM12]. Alle diese Algorithmen basieren auf einer iterativen Verfeinerung und kombinieren ein billiges, "grobes" numerisches mit einem teuren, dafür aber "feinen" numerischen Verfahren. Die sequentielle Lösung, die durch das grobe Verfahren entstanden ist, wird mit Hilfe paralleler Ausführung eines feinen Verfahrens iterativ verbessert.

Einen guten Überblick über unterschiedliche parallele Lösungsverfahren findet man in [Bur95]. Als nächstes wird eine kurze Übersicht paralleler RK-Verfahren gegeben.

Das Parallelitätspotential von RK-Verfahren hängt von deren Berechnungsstruktur ab. Aufgrund der Abhängigkeit numerischer Verfahren zur Lösung von DGL von der Zeitkomponente ist die Parallelisierung auf Basis von Zeitschritten schwer und geht meistens zur Lasten von Genauigkeitsanforderungen. Bevor man eine neue Approximation $\boldsymbol{\eta}_{\kappa+1}$

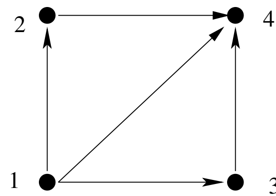


Abbildung 3.2.: Das Produktionsgraph zum Butcher-Tableau (3.25) [HNW09a].

berechnen kann, muss man in der Regel entweder die zuletzt berechnete Approximation $\boldsymbol{\eta}_j$ kennen oder extrapolieren. Für eine spezielle Klasse von RK-Verfahren, nämlich PC-Verfahren vom RK-Typ, ist die Parallelisierung bezüglich der Zeitschritte möglich, wie in [vdHSvdV94, vdHSvdV95] gezeigt wird. Diese Ansätze erlauben aber nur eine eingeschränkte Schrittweitenkontrolle. Die Parallelisierung bezüglich des Systems ist für alle RK-Verfahren möglich. Dagegen ist die Parallelisierung bezüglich der Methode nur auf RK-Verfahren mit spezieller Berechnungsstruktur anwendbar.

3.5.1. Parallele explizite RK-Verfahren

Die expliziten RK-Verfahren sind nur eingeschränkt bezüglich der parallelen Berechnung der Stufen parallelisierbar.

Die Anzahl der sequentiellen Stufen q eines Verfahrens ist die Länge des längsten Pfades in dem Produktionsgraphen. Für jedes RK-Verfahren kann ein Produktionsgraph konstruiert werden. Als Beispiel dient das folgende Butcher-Tableau eines parallelen Verfahrens [HNW09a]:

$$\begin{array}{c|cccc} 0 & & & & \\ x & x & & & \\ x & x & 0 & & \\ x & x & x & x & \\ \hline & x & x & x & x \end{array} \tag{3.25}$$

Um einen Produktionsgraphen zu erhalten, zeichnet man für alle Einträge $a_{li} \neq 0$ in dem Butcher-Tableau eine Kante vom Knoten “i” zu Knoten “l” . Der Produktionsgraph zum Tableau (3.25) ist in Abbildung 3.2 gegeben. Der Produktionsgraph besitzt 3 sequentielle Stufen. Die Stufen 2 und 3 sind unabhängig voneinander und können deshalb parallel berechnet werden. Verfahren, bei denen die Anzahl der sequentiellen Stufen q gleich der Ordnung p des Verfahrens ist, d. h. $p = q$, werden als **optimal** bezeichnet. Diese Verfahren sind optimal, weil sie eine höhere Genauigkeit bieten und eine maximal mögliche Parallelität bezüglich der parallelen Stufenberechnung erreichen.

Der folgende Satz von Jackson und Nørsett (1992) schränkt das Parallelitätspotential von expliziten RK-Verfahren bezüglich der parallelen Berechnung der Stufen ein [HNW09a]:

Satz 3.5.1. *Für die Ordnung p eines expliziten RK-Verfahrens mit q sequentiellen Stufen*

gilt die Bedingung

$$p \leq q$$

für jede beliebige Anzahl verfügbarer Prozessoren.

Aus Satz 3.5.1 folgt, dass bei einem optimalen Verfahren der Ordnung p maximal $s-p$ Stufen parallelisierbar sind [Kor07]. Eine Möglichkeit zur Konstruktion von optimalen Verfahren mittels Fixpunktiteration haben Nørsett und Simonsen [NS89] und van der Houwen u. Sommermeijer [vdHS90a] gezeigt. Sie haben vorgeschlagen, implizite RK-Verfahren als Basisverfahren für PC-Iteration zu nutzen. Bei impliziten RK-Verfahren ist die Berechnung der Stufenvektoren durch ein nicht-lineares GLS der Größe $s \cdot n$ gegeben. Verwendet man eine Fixpunktiteration zur Lösung des nicht-linearen GLS, so entsteht ein **explizites PC-Verfahren vom RK-Typ**. Explizite PC-Verfahren von RK-Type führen, ausgehend von einem Startwert, z. B. trivialen Prädiktor

$$Y_l^{(0)} = \boldsymbol{\eta}_\kappa \quad \text{für} \quad l = 1, \ldots, s, \tag{3.26}$$

eine Fixpunktiteration mit einer festen Anzahl m von Iterationen durch, um das resultierende nicht-lineare GLS zu lösen:

$$\begin{aligned} \mathbf{Y}_l^{(k)} &= \boldsymbol{\eta}_\kappa + h_\kappa \sum_{i=1}^{s} a_{li} \mathbf{F}_i^{(k-1)}, \\ \mathbf{F}_i^{(k-1)} &= \mathbf{f}\left(t_\kappa + c_i h_\kappa, \mathbf{Y}_i^{(k-1)}\right), \\ l &= 1, \ldots, s, \quad k = 1, \ldots, m. \end{aligned} \tag{3.27}$$

Die neue Approximation an der Stelle $t_{\kappa+1}$ wird dann wie folgt berechnet:

$$\boldsymbol{\eta}_{\kappa+1} = \boldsymbol{\eta}_\kappa + h_\kappa \sum_{i=1}^{s} b_l \mathbf{F}_l^{(m)}. \tag{3.28}$$

Zusätzlich berechnet man eine zweite Approximation durch Verwendung der vorletzten Approximationen der Stufenvektoren:

$$\hat{\boldsymbol{\eta}}_{\kappa+1} = \boldsymbol{\eta}_\kappa + h_\kappa \sum_{i=1}^{s} b_l \mathbf{F}_l^{(m-1)} \tag{3.29}$$

Explizite PC-Verfahren vom RK-Typ der Form (3.27) werden in der Literatur auch **(parallel) iterierte RK-Verfahren** (vgl. [vdHS90b]) genannt. Die Vektoren $\boldsymbol{\eta}_{\kappa+1}$ und $\hat{\boldsymbol{\eta}}_{\kappa+1}$ werden zur Schrittweitensteuerung verwendet. Für die Berechnung beider Approximationen $\boldsymbol{\eta}_{\kappa+1}$, $\hat{\boldsymbol{\eta}}_{\kappa+1}$ werden die gleichen Koeffizienten b_i des Gewichtungsvektors verwendet. Die Verfahrenskoeffizienten a_{li}, b_i, c_i, sowie die Anzahl der Iterationen m sind durch das zugrunde liegende implizite RK-Verfahren gegeben.

Das PC-Verfahren vom Runge-Kutta-Typ der Form (3.27) kann als explizites RK-

Verfahren durch das folgende Butcher-Tableau dargestellt werden [NS89]:

$$\begin{array}{c|cccccc} 0 & 0 & & & & & \\ c & A & 0 & & & & \\ c & 0 & A & 0 & & & \\ c & 0 & 0 & A & 0 & & \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \\ c & 0 & \dots & \dots & 0 & A & 0 \\ \hline & 0^T & \dots & \dots & \dots & 0^T & b^T \end{array} \tag{3.30}$$

Ist p die Ordnung des zugrunde liegenden s-stufigen RK-Verfahrens, so hat das entstehende $(m+1)\cdot s$-stufige explizite PC-Verfahren vom RK-Typ die Ordnung $\min(p, m+1)$ (für einen Beweis siehe [NS89]) [RR00]. Deshalb wird die Anzahl der Iteration m bei PC-Verfahren vom RK-Typ in der Regel auf $m = p - 1$ gesetzt [RR00].

PC-Verfahren vom RK-Typ der Form (3.27) bieten ein zusätzliches Parallelitätspotential. Da bei diesem Verfahren die Stufenvektoren $\mathbf{Y}_1^k, \dots, \mathbf{Y}_s^k$ innerhalb einer Iteration k (auch Korrektorschritt genannt) unabhängig voneinander sind, können diese parallel berechnet werden. Damit ist bei expliziten PC-Verfahren vom RK-Typ nicht nur die Parallelisierung bezüglich des Systems, sondern auch bezüglich der Methode (parallele Berechnung der Stufenvektoren) möglich.

Der folgende Satz gibt die untere Schranke für die minimale Anzahl der Prozessoren bezüglich der Methodenparallelität zur Implementierung eines optimalen PC-Verfahrens vom RK-Typ an:

Satz 3.5.2. *Explizite PC-Verfahren vom RK-Typ der Form (3.27) mit der Ordnung* $\min(p, m+1)$ *und* $(m+1)\cdot s$ *Stufen sind optimal auf* p *Prozessoren, wenn* $m \le p-1$ *[vdHS90b].*

Es existieren optimale explizite PC-Verfahren vom RK-Typ auf $(p+1)/2$ Prozessoren, entsprechende Verfahren verwenden Radau-Verfahren der Ordnung p als Basisverfahren [vdHS90b].

In der Arbeit von Rauber und Rünger [RR96] wurde das Parallelitätspotential von expliziten PC-Verfahren vom RK-Typ untersucht. Es wurden unterschiedliche parallele Implementierungen für Rechner mit verteiltem Speicher erstellt. Die Implementierungsvarianten nutzen beide Arten der Parallelität bezüglich der Methode und des Systems aus und unterscheiden sich in der Verteilung der Daten. In [RR96] wurde ein Laufzeitvorhersagemodell präsentiert, mit dem man für einzelne Implementierungen Berechnungs- und Kommunikationszeiten abschätzen kann. Ansätze zur Implementierung expliziter PC-Verfahren vom RK-Typ für Rechner mit gemeinsamem Speicher wurden von Korch und Rauber in [KR07b] vorgestellt. Die Parallelisierung erfolgte bezüglich des Systems. Das Skalierbarkeitsverhalten der Implementierungen wurde in Abhängigkeit von Speicherzugriffslokalität untersucht und es wurden Ansätze zur Optimierung des Lokalitätsverhaltens entwickelt.

Ansätze zur parallelen Ausführung eingebetteter RK-Verfahren, die das Parallelitätspotential bezüglich des Systems ausnutzen, findet man in [RR04, Kor07, KR06]. In

[KR11] wurde eine parallele Pipelining-Implementierung eingebetteter RK-Verfahren mit reduziertem Speicherbedarf präsentiert. Im Gegensatz zu den Arbeiten [KCL00, BBB04, Ruu05] gibt dieser Ansatz keine Einschränkung bezüglich der Wahl von Verfahrenskoeffizienten vor und kann deshalb auf alle eingebetteten RK-Verfahren angewendet werden.

3.5.2. Parallele implizite RK-Verfahren

Implizite RK-Verfahren haben den Nachteil, dass in jedem Zeitschritt ein nichtlineares GLS gelöst werden muss. Zur Lösung des Differentialgleichungssystems mit n Gleichungen und einem s-stufigen RK-Verfahren muss ein nichtlineares GLS der Dimension $s \cdot n$ gelöst werden. Deshalb basieren die meisten parallelen impliziten RK-Verfahren auf parallelen Implementierungen des Newton-Verfahrens. Da bei klassischen impliziten RK-Verfahren das Berechnen von Stufenvektoren nicht entkoppelt von der Lösung eines nichtlinearen GLS stattfinden kann, ergibt sich für das Newton-Verfahren die Laufzeit von $O(s^3n^3)$, wenn ein direktes Verfahren wie z. B. Gauß zur Lösung des entstehenden linearen GLS verwendet wird [RR00]. Der Berechnungsaufwand des Newton-Verfahrens lässt sich jedoch für eine spezielle Klasse impliziter RK-Verfahren, die sog. **diagonal impliziten iterierten RK-Verfahren (DIRK)**, deutlich reduzieren. Die DIRK-Verfahren gehen auf die Arbeit [Ale77] von Alexander (1977) zurück und sind charakterisiert durch eine Verfahrensmatrix A, die eine untere Dreiecksmatrix ist. Eine Verfahrensmatrix ist eine untere Dreiecksmatrix, wenn alle Einträge oberhalb der Hauptdiagonale 0 sind. Durch die spezielle Berechnungsstruktur von DIRK-Verfahren reduziert sich der Berechnungsaufwand des Newton-Verfahrens, da in jedem Iterationsschritt m anstatt eines GLS der Größe $s \cdot n$ nur s entkoppelte Gleichungen der Größe n gelöst werden müssen. Wird ein Newton-Verfahren verwendet, so ist die Laufzeit eines Zeitschrittes der DIRK-Verfahren $O(s \cdot n^3 \cdot m)$ [RR00]. Der Vorteil von DIRK-Verfahren gegenüber anderen impliziten RK-Verfahren ist die Möglichkeit der parallelen Berechnung von Stufenvektoren (s-unabhängige, nicht-lineare Gleichungen) innerhalb eines Iterationsschrittes. Arbeiten zu parallelen DIRK-Verfahren findet man u. a. in [vdHSC92, RR95, RR99, IN90, RR00].

4. Automatisches Tuning von Software

In diesem Kapitel werden die wichtigsten Grundlagen zum **automatischen Tuning von Software (Autotuning)** dargestellt. Die Notwendigkeit des automatischen Optimierens von Software wird dadurch motiviert, dass die aktuellen Entwicklungen im Bereich der Hardwarearchitektur beschrieben und die sich daraus ergebenden Herausforderungen für die Softwareentwicklung erörtert werden. Es folgt die Definition des Autotuning-Problems und die anschließende Unterteilung der Autotuning-Ansätze nach dem Zeitpunkt der Evaluierung von Implementierungsvarianten und Bestimmung von Parametern. Des Weiteren werden die Strategien zur Traversierung des Suchraumes beschrieben und die allgemeinen Herausforderungen und Probleme bei der Entwicklung eines Autotuning-Ansatzes skizziert.

4.1. Aktuelle Entwicklungen im Bereich der Rechnerarchitektur

Laut Mooreschem Gesetz verdoppelt sich die Anzahl der Schaltelemente (Transistoren) auf derselben Fläche eines Chips bei gleichbleibenden Kosten in etwa alle 18 Monate [Moo65] [1]. Moore's Gesetz ist kein Naturgesetz, sondern eine empirische Beobachtung, die Gordon Moore als Chef der Halbleiterfirma Intel im Jahr 1969 gemacht hat. Dass das Gesetz von Moore bis heute seine Gültigkeit behielt, ist vor allem der Innovation im Bereich der Halbleitertechnologie zu verdanken. In der bisherigen Entwicklung von Prozessoren konnte die Anzahl der Transistoren auf einem Chip durch die Verringerung der Strukturbreite von Transistoren immer weiter erhöht werden. Die technischen Probleme konnten durch neue Verfahren, wie z. B. neue fotolithographische Techniken, immer wieder überwunden werden. Da die Bauteile eines Chips aber ständig kleiner werden, wird es für die Chip-Hersteller immer schwieriger, dem Mooreschen Gesetz zu folgen. Des Weiteren mehren sich die Stimmen, eine davon ist die des ehemaligen Intel-Mitarbeiters Robert Colwells [Cro13], die bis zum Jahr 2020 das Ende des Mooreschen Gesetzes prophezeien. Als Grund dafür nennt Colwell vor allem die Explosion der Kosten bei der Fertigung von Chips mit immer kleineren Strukturen [Cro13]. Die Entwicklung von Prozessoren mit Transistoren stets geringerer Strukturbreite lässt sich am Beispiel der Firma Intel beobachten. Seit 2007 verwendet die Firma Intel für die Entwicklung von Prozessoren die so genannte **"Tick-Tock-Strategie"**. Auf eine Strukturverkleinerung ("Tick") kommt die Einführung einer neuen Prozessorarchitektur ("Tock"). Der aktuelle Prozessor von Intel, der "Haswell"-Prozessor mit einer Fertigungstechnologie von 22 Nanometern, ist durch die Veränderung der Mikroarchitektur entstanden. Bis Anfang 2015 soll sein Nachfolger erscheinen, der "Broadwell"-Prozessor, der mit einer 14 Nanometer Fertigungstechnologie

[1] Die Anpassung der Abschätzung von 24 auf 18 Monate wurde durch Moores Kollegen David House aufgestellt.

arbeiten soll. Die Haswell-Prozessoren bestehen aus rund 1,5 Milliarden Transistoren, der 14-Nanometer-"Broadwell“-Chip soll über 4 Milliarden Transistoren verfügen [Man14]. Bis 2019 will Intel in mehreren Zwischenschritten die Transistorbreite von 5 Nanometer erreichen. Weiter dürfte die Transistorbreite auf Grund der physikalischen Grenzen kaum noch schrumpfen.

Eine Verdopplung der Transistoranzahl geht nicht unbedingt einher mit der Verdopplung der Leistung von Prozessoren einher, da sich die Taktrate eines Prozessors aufgrund von steigender Wärmeentwicklung nicht beliebig steigern lässt. Anstatt die Taktrate immer weiter zu erhöhen, ist man zu Herstellung von **Mehrkernprozessoren** übergegangen. Die Mehrkernprozessoren bestehen aus mehreren CPUs auf einem Chip. Die Leistungsfähigkeit von Rechnersystemen lässt sich noch weiter durch den Bau von **Multiprozessorsystemen** steigern. Multiprozessorsysteme sind Computer mit mehreren Prozessoren, welche auch Mehrkernprozessoren sein können, die über ein Netzwerk miteinander verbunden sind. Ein Beispiel dafür sind Clustersysteme. Die Anzahl der Prozessoren in Clustersystemen sowie auch der CPU-Kerne pro Prozessor nimmt ständig zu und es ist zu erwarten, dass sich dieser Trend in der Zukunft weiter fortsetzt, was zu **Supercomputern** der Exascale-Klasse mit Millionen von Prozessorkernen führen wird [CKLV11]. Für das Jahr 2019 hat der Prozessorhersteller Intel bereits den ersten Supercomputer angekündigt, der die Rechenleistung eines ExaFLOPS (10^{18} Floating Point Operations Per Second (FLOPS)) überschreiten soll [Ris13]. Einen Überblick über die schnellsten Supercomputersysteme nach Rechenleistung liefert die Top-500-Liste [top14]. Eine andere Liste, Green-500 [Com14], bewertet die Supercomputer nach der Rechenleistung und dem Energiebedarf. Bei dieser wird als das Maß für die Bewertung der Supercomputersysteme die Anzahl der FLOPS pro Watt angenommen. Der aktuelle Spitzenreiter in der Top-500-Liste ist das chinesische Rechnersystem Tianhe-2. Dieses besteht aus 16.000 Rechenknoten mit 3,12 Millionen Kernen und bietet die Rechenleistung von 33,86 PetaFLOPS ($33,86^{15}$ FLOPS) [top14]. Die Systeme in der Top-500-Liste basieren auf unterschiedlichen Prozessorarchitekturen, sind aber alle Parallelrechner, bei denen sich mehrere Recheneinheiten die Arbeit teilen. Die meisten Systeme in der Top-500-Liste sind Clustersysteme, die aus mehreren Multicore-Standardprozessoren von Intel, AMD, IBM u. a. aufgebaut sind. Andere Systeme in dieser Liste sind heterogene Systeme. Diese kombinieren Standardprozessoren mit Zusatzprozessoren aus Grafikkarten. Das erstplatzierte System Tianhe-2, und die Nummer 7 in der Top-500-Liste, Stampede, setzen Xeon-Phi-Prozessoren von Intel ein [top14]. Dagegen nutzen die Nummer 2, Titan, und die Nummer 6, Piz Daint, Nvidia-GPUs zur Beschleunigung von Berechnungen [top14]. In der Top-500-Liste findet man auf den vorderen Plätzen ebenfalls die BlueGene/Q-Systeme von IBM.

Die parallelen Rechnerarchitekturen lassen sich je nach physikalischer Organisation des Speichers in Rechner mit verteiltem und in Rechner mit gemeinsamem Speicher unterteilten [RR12a]. Ein Rechner, bei dem jeder Prozessor physikalisch über einen eigenen Speicher verfügt, wird als Rechner mit **verteiltem Speicher (engl. distributed memory machine)** bezeichnet. Der einzelne Prozessor darf direkt nur auf seinen eigenen lokalen Speicher zugreifen. Wenn ein Prozessor zur Datenverarbeitung die Daten

aus dem Speicher eines anderen Prozessors benötigt, so werden diese Daten durch **das Versenden von Nachrichten (engl. message passing)** über das Verbindungsnetzwerk zur Verfügung gestellt. Bei einem Rechner mit **gemeinsam genutztem Speicher (engl. shared memory machine)** teilen sich mehrere Prozessoren eines Multiprozessorsystems den gemeinsamen (lokalen oder globalen) Speicher. Der gemeinsame Speicher besteht in der Regel aus mehreren Speichermodulen. Diese bilden zusammen einen gemeinsamen Adressraum, auf den alle Prozessoren über ein Verbindungsnetzwerk zugreifen können. Der Austausch von Daten zwischen einzelnen Prozessoren auf einem Rechner mit gemeinsamem Speicher kann durch die Verwendung von **gemeinsamen Variablen (engl. shared variables)** erfolgen. Ebenso kann der Austausch von Daten wie bei Rechnern mit verteiltem Speicher durch das Versenden von Nachrichten realisiert werden.

Die früheren Multiprozessorsysteme mit gemeinsam genutztem Speicher hatten einen einzigen globalen Speicher, auf den alle Prozessoren über einen gemeinsamen Bus und mit gleicher Speicherzugriffslatenz zugreifen konnten. Diese Art von Speicher-Architektur wird **UMA-Architektur (Uniform Memory Access)** genannt. Da bei dieser Form der Speicher-Architektur alle Speicherzugriffe über einen gemeinsamen Bus erfolgen, kann dieser schnell zum "Flaschenhals" werden. Daher sind derartige Rechner nur eingeschränkt erweiterbar und bestehen oft nur aus wenigen Prozessoren. Die modernen Mulitiprozessorsysteme mit gemeinsam genutztem Speicher sind meistens so konzipiert, dass der Zugriff auf lokale und entfernte Speichermodule mit unterschiedlicher Geschwindigkeit erfolgt. Dieses Konzept der Speicher-Architektur wird als **nicht-uniformer Speicherzugriff (engl. NUMA (NON-Uniform Memory Access))** bezeichnet. Der Zugriff auf einen dem Prozessor physikalisch nah liegenden Speichermodul hat eine kürzere Latenzzeit als der Zugriff auf ein entferntes Speichermodul des anderen Prozessors. Die NUMA-Rechner verwenden für die Kommunikation zwischen den Prozessoren anstatt der Busse entweder **bidirektionale Hochgeschwindigkeitsverbindungen**, wie z. B. das **HyperTransport (HT)** von AMD, oder einen **Routing-Mechanismus**, wie das **QuickPath Interconnect (QPI)** von Intel.

Die Zugriffszeit auf entfernte Speichermodule kann durch schnelle Speicher, auch **Caches** genannt (siehe Kapitel 6.1), verborgen werden. Moderne CPUs verfügen über mehrere Caches, in der Regel zwei oder drei, unterschiedlicher Kapazität und Zugriffszeit. Bei Multiprozessorsystemen mit lokalen Caches muss die **Cache-Kohärenz** hergestellt werden, weil es ansonsten zu Problemen kommen kann [RR12a]. Ein Speichersystem besitzt die Eigenschaft der **Cache-Kohärenz**, wenn ein Lesezugriff immer den Wert des letzten Schreibzugriffs auf die entsprechende Speicherzelle liefert [Ung10]. Die Cache-Kohärenz kann mit Hilfe von **Cache-Kohärenz-Protokollen** sichergestellt werden. Werden bei einem NUMA-System kohärente Caches verwendet, so bezeichnet man diese Systeme als **cache coherent NUMA (ccNUMA)** [Tan09, RR12a].

Das aktuell größte Rechnersystem mit gemeinsam genutztem Speicher ist das UV 2000 System des Herstellers SGI, das den Beinamen "The Big Brain Computer" trägt [Cor14]. In der Maximalkonfiguration kann dieses System bis zu 2048 Prozessorkerne (4,096 Threads) enthalten, die alle auf den gemeinsamen Speicher (bis zu 64 Terabyte)

zugreifen. Das SGI UV 2000 ist ein ccNUMA-System, das mit den Prozessoren der Produktfamilie Intel Xeon E5 ausgestattet ist.

Um die Wartezeit auf Speicherzugriffe zu überdecken, können moderne Prozessoren mehrere Threads gleichzeitig ausführen. Diese Technik wird als **simultanes Multithreading (SMT)** bezeichnet. Durch die gleichzeitige Ausführung mehrerer Threads können die Ressourcen eines Prozessors besser ausgenutzt werden.

Die aktuellen Prozessoren bieten ferner unterschiedliche Technologien zur Änderung der Taktfrequenz einzelner Prozessorkerne in Abhängigkeit von der Prozessorlast. Zu diesen Technologien gehört die SpeedStep-Technologie [Int14a] von Intel. Diese wurde ursprünglich dazu entwickelt, den Energieverbrauch von Notebooks zu reduzieren. Mit Hilfe der SpeedStep-Technologie kann der Prozessor die Taktrate entsprechend der CPU-Auslastung entweder erhöhen oder absenken. Die **Turbo-Boost** [Int14b] ist eine Technik zur automatischen Übertakten von Prozessoren. Werden durch eine Anwendung nicht alle CPU-Kerne ausgelastet, so kann die Taktfrequenz einzelner Kerne erhöht werden, während andere Kerne schlafen gelegt werden. Die Gesamtwärme eines Prozessors darf dabei nicht überschritten werden.

4.2. Herausforderungen in der Softwareentwicklung

Durch den vorherigen Exkurs über die Trends in der Hardwareentwicklung wird deutlich, dass die Komplexität moderner CPUs und Multiprozessorsysteme immer weiter zunimmt. Die Multiprozessorsysteme werden zunehmend heterogen und zeichnen sich durch immer tiefere Speicherhierarchien, hierarchische Verbindungsnetzwerke und die Verfügbarkeit von SIMD-Vektorinstruktionen aus. Die Anforderungen an die Software wachsen sehr schnell, und damit eine Software nicht veraltet und auch auf der aktuellen Prozessorarchitektur effizient läuft, muss diese in regelmäßigen Abständen an die Entwicklung der Hardware angepasst werden. Insbesondere für die Anwendungen im Bereich der Simulation naturwissenschaftlicher Phänomene ist die Effizienz von Programmen von entscheidender Bedeutung [RR12b]. Solche Anwendungen sind Computersimulationen mit sehr hohen Berechnungsaufwand, wie z. B. Klima- und Wettersimulationen [RR12b]. Die Performance von sequentiellen Programmen lässt sich entweder durch eine Optimierung des Programmcodes oder durch eine **Parallelisierung** steigern. Die Effizienz eines parallelen Programms hängt sehr stark von den Eigenschaften der Zielplattform ab, wie z. B. der Multicore-Architektur, der Cachearchitektur sowie der Topologie und Bandbreite des Speichers. Um eine gute Performance zu erreichen, muss deshalb der (parallele) Programmcode für die Zielplattform optimiert werden. Da aber neue Prozessorgenerationen in immer kürzeren Abständen auf den Markt kommen und die Komplexität der Hardware immer weiter zunimmt, ist das manuelle Optimieren von Programmcode, auch **manuelles Tuning** genannt, unpraktikabel. Das manuelle Tuning verlangt vom Programmierer viel Zeit und ist daher sehr kostenintensiv. Mehr noch: Ein Programmcode, der für eine bestimmte Architektur optimiert worden ist und auf dieser sehr gute Performance liefert, kann auf einer Anderen eine unterdurchschnittliche Performance aufweisen. Der Code muss deshalb für jede neue Architektur erneut optimiert werden.

Bei manchen Anwendungen hängt die Performance auch stark von den Eingabedaten ab. Dieser Effekt erschwert das manuelle Tuning zusätzlich.

Um den Aufwand des manuellen Tunings zu vermeiden, werden zur Codeoptimierung vor allem unterschiedliche Compiler-Techniken verwendet [AK02, Muc97]. Die Schwierigkeit für Compiler besteht aber darin, dass diese nicht über das Wissen eines Programmier-Experten verfügen. Compiler sind nicht in der Lage, Veränderungen in den Datenstrukturen vorzunehmen, und können nur dann Optimierungstechniken anwenden, wenn es die Daten- und Kontrollabhängigkeiten im Programm erlauben (mehr in Kapitel 6.4). Compiler besitzen auch keine oder nur geringe Kenntnisse über die Eingabe des Programms. Seit einiger Zeit geht daher vor allem im wissenschaftlichen Bereich der Trend bei der Programmoptimierung zum **automatischen Tuning von Software (Autotuning)**.

4.3. Das automatische Tuning von Software

Beim automatischen Software-Tuning soll das Programm auf das vorgegebene Ziel hin automatisch optimiert werden. Die Ziele zur Optimierung können u. a. die Ausführungszeit, der Energieverbrauch oder auch ein Kompromiss zwischen den beiden sein, wie z. B. das Energy-Delay-Produkt [HIG94]. Soll das Programm hinsichtlich mehrerer Ziele gleichzeitig optimiert werden, so spricht man vom **mehrkriteriellen Autotuning (engl. multi-objective Tuning)**. Beim mehrkriteriellen Tuning können sich die zu erreichende Ziele teilweise widersprechen, sodass ein Optimum nicht durch eine Lösung zu erreichen ist. Bei mehrkriteriellem Autotuning wird daher nicht nach einer guten Lösung, sondern nach einer Menge von nicht-dominierten Lösungen gesucht, die einen Kompromiss zwischen unterschiedlichen Optimierungszielen darstellen. In dieser Arbeit wird das automatische Tuning bezüglich eines Optimierungsziels betrachtet.

Das Autotuning ist ein automatischer Prozess, der meistens aus drei Phasen besteht [Vud11]:

- **Definition des Suchraumes**. Es wird festgelegt, welche Programmparameter, Algorithmen und Optimierungstechniken während des Autotunings evaluiert werden sollen.

- **Generierung von Implementierungsvarianten**. Zum Suchraum werden entsprechende Implementierungsvarianten generiert. Die Programmvarianten, auch **Implementierungskandidaten genannt (engl. candidate implementations)**, bilden zusammen den so genannten **Kandidatenpool (engl. pool of candidates)**.

- **Evaluierung der Implementierungsvarianten aus dem Kandidatenpool**. Es wird nach dem Implementierungskandidaten gesucht, der das beste Ergebnis in Hinblick auf das Optimierungsziel (z. B. schnellste Laufzeit, geringster Energiebedarf) bietet.

Vom Autotuning spricht man bereits, sobald eine dieser Phasen automatisch abläuft [Vud11].

4.3.1. Offline- versus Online-Autotuning

Je nachdem, zu welchem Zeitpunkt die Evaluierung von Implementierungsvarianten und Parametern geschieht, unterscheidet man zwischen **Online-Autotuning** und **Offline-Autotuning**. Das Offline-Autotuning findet vor dem Ausführen des Programms statt, meistens zur Installations- oder zur Compile-Zeit. Das Offline-Autotuning wird oft bei den Anwendungen eingesetzt, bei welchen die beste Implementierungsvariante von den Eigenschaften der Zielplattform und nicht von dem zu lösenden Problem abhängt. Offline Autotuning wurde erfolgreich bei vielen Anwendungen aus dem Gebiet der linearen Algebra angewandt [BACD97, WD97] (siehe Kapitel 5).

Für einige Anwendungen jedoch, wie auch für die in dieser Arbeit betrachteten Solver zur Lösung von AWP von ODEs, reicht das Installations- oder Compile-Zeit-Tuning nicht aus, um die gewünschte Performance zu erzielen. Bei den ODE-Solvern hängt die erreichte Performance nicht nur von den Eigenschaften der Zielplattform ab, sondern wird im starken Maße von erst zur Laufzeit verfügbaren Informationen, wie z. B. dem zu lösenden ODE-Problem und dessen Größe, beeinflusst. Das führt dazu, dass auf der gleichen Zielplattform, aber für unterschiedliche Eingaben unterschiedliche Programmvarianten die schnellsten sein können. Da die Performance beeinflussende Eigenschaften des Problems, wie z. B. die Größe und das Zugriffsmuster eines ODE-Problems, erst zur Laufzeit bekannt sind, muss auch das Autotuning zur Laufzeit durchgeführt werden. Findet der Tuning-Prozess zur Laufzeit statt, so spricht man vom Online-Autotuning.

Die Suche nach der besten Programmvariante ist generell mit einem Zeitaufwand verbunden. Beim Offline-Autotuning ist die Zeit für das Autotuning in der Installations- und Compilierungsphase verborgen. Das Tuning geschieht nur einmal, nämlich wenn das Programm auf der Zielarchitektur installiert und compiliert wird. Deshalb kann beim Offline-Autotuning, je nach Anwendung, ein Aufwand von mehreren Stunden bis Tagen vertretbar sein. Beispielsweise sind bei der Offline-Autotuning-Bibliothek ATLAS [WD97], je nach verwendeten Compiler und Architektur, Zeiten von mehreren Stunden typisch [ATL09]. Beim Tuning zur Laufzeit wären solche lange Zeiten inakzeptabel. Beim Online-Autotuning sollte die Zeit, die für das Tuning aufgewendet wird, durch die Zeit der anschließenden Berechnung mit der besten Variante oder durch die mehrmalige Ausführung der Berechnung weitgehend kompensiert werden. Viele Autotuning-Bibliotheken kombinieren beide Ansätze: das Offline- mit dem Online-Autotuning.

4.3.2. Strategien zur Traversierung des Suchraumes

Die Autotuning-Ansätze lassen sich anhand der Strategie zur Traversierung des Suchraumes in zwei Kategorien unterteilen: **modellbasierte** und **empirische** Ansätze. Modellbasierte Autotuner verwenden Modelle, z. B. zur Laufzeitvorhersage. Die empirischen Autotuner treffen ihre Entscheidungen auf der Suche nach der besten Implementierungsvariante basierend auf empirischen Daten, wie z. B. der Laufzeitmessung. Die Idee der

modellbasierten Optimierung kommt aus dem Compilerbereich [LDT09]. Im Compilerbereich wurden unterschiedliche Optimierungstechniken entwickelt, wie z. B. Loop-Unrolling oder Loop-Tiling, um die Performance eines Programmcodes auf modernen Architekturen zu verbessern. Um geeignete Parameter, wie z. B. die Blockgröße beim Loop-Tiling oder den Aufrollfaktor beim Loop-Unrolling, für diese Optimierungstechniken zu bestimmen, verwenden Compiler oft analytische Modelle. Diese sind meistens nur eine vereinfachte Abstraktion der wesentlichen Eigenschaften der Zielplattform und der Implementierung [LDT09]. Im Kontext der ATLAS-Bibliothek [YLR+05] wurde gezeigt, dass die Auswahl geeigneter Blockgrößen und Aufrollfaktoren mit Hilfe eines analytischen Modells sehr gut funktionieren kann. Für viele hochspezialisierte Anwendungen sind jedoch analytische Modelle aufgrund der hohen Abstraktion zu ungenau. Ein weiterer Nachteil ist überdies, dass präzise analytische Modelle sehr schwer zu konstruieren sind.

Eine Alternative zur modellbasierten Optimierung bietet die empirische Optimierung. Dabei werden mehrere Varianten eines Programmcodes erzeugt, die anschließend auf der Zielplattform ausgeführt werden. Unter allen Programmvarianten wird dann automatisch diejenige Variante ausgewählt, die das beste Ergebnis in Bezug auf das jeweilige Optimierungsziel liefert. Soll beispielsweise die Ausführungszeit eines Programms optimiert werden, so wird anhand der Laufzeiten die schnellste Programmvariante ausgewählt. Die Zeit für das empirische Autotuning wächst exponentiell mit der Anzahl zu evaluierender Varianten und Parameter. Wie gut das empirischen Tuning funktioniert, hängt deshalb stark von den verwendeten Suchalgorithmen ab. Da die Anzahl zu testender Programmvarianten sehr groß sein kann, ist die Anwendung einer erschöpfenden Suche meistens unpraktikabel. Um die Zeit für das Autotuning möglichst gering zu halten, wird oft eine **modellgeführte (engl. model-guided)** Suche verwendet [CCH05, SSF+12]. Bei dieser werden die Ansätze der empirischen und modellbasierten Suche miteinander kombiniert. Das analytische Modell wird verwendet, um die Anzahl der Implementierungsvarianten für die anschließende empirische Suche einzuschränken. Die empirische Suche kann zudem durch die Evaluierung synthetischer Testbeispiele oder eine Heuristik unterstützt werden.

Das Autotuning-Problem kann mathematisch als ein kombinatorisches Optimierungsproblem formuliert werden [BWH11, Chr11a]. Ist kein Modell für die Zielfunktion gegeben, so ist beim empirischen Autotuning der Verlauf der Zielfunktion nicht bekannt. Stattdessen entspricht die Auswertung der Zielfunktion an einem bestimmten Punkt dem Messergebnis eines Experiments, genauer: dem Ausführen einer Programmvariante mit den zu testenden Parametern und dem Sammeln von empirischen Daten, beispielsweise dem Messen der Laufzeit [Chr11a]. Da die Ableitung der Zielfunktion nicht bestimmt werden kann, kommen für die empirische Suche nur **ableitungsfreie (engl. derivation-free)** Suchverfahren in Frage. Die ableitungsfreien Suchverfahren verwenden keine Ableitungen der Zielfunktion. Bei ableitungsfreien Suchverfahren wird die Suchrichtung durch die Auswertung der Zielfunktion an einem Punkt und dem Vergleich mit anderen Punkten in der Nachbarschaft bestimmt. Zu den ableitungsfreien Suchverfahren gehören direkte Suchverfahren, wie z. B. die erschöpfende Suche und das Simplex-Verfahren

von Nelder und Mead [NM65]), und metaheuristische Suchverfahren, wie das Simulated Annealing [KGV83], genetische [Gol89] und evolutionäre Algorithmen [Hol92], und Partikelschwarmoptimierung [Ken10]. Eine Übersicht ableitungsfreier Suchverfahren ist in [RS13] gegeben. Ein konzeptueller Vergleich von Metaheuristiken ist in [BR03] zu finden. In [KKO00] wurde die Effektivität unterschiedlicher Algorithmen, u. a. der zufälligen Suche, des Simulated Annealings und eines genetischen Algorithmus für die automatische Auswahl von Aufrollfaktoren und Blockgrößen untersucht. Eine ähnliche Studie mit dem Vergleich unterschiedlicher Suchalgorithmen ist in [SYD08] zu finden. In beiden Publikationen [KKO00, SYD08] war die zufällige Suche die effizienteste Methode. Der Grund dafür war, dass es im Suchraum viele Punkte mit guter Effizienz gab, deren Performance maximal 5 % vom Optimum entfernt war. In [BWH13] wurden anhand der Beispiele aus der SPAPT Test-Suite [BWN12] die globalen und lokalen Suchalgorithmen miteinander verglichen. Die Ergebnisse in [BWH13] beschränken sich auf Tuning-Parameter, deren Wertebereich ganzzahlig ist. Das Fazit in [BWH13] ist, dass die lokalen Suchalgorithmen gute Ergebnisse in kurzer Zeit liefern, wenn der Startpunkt der Suche gut gewählt ist. Beim Online-Autotuning ist wichtig, geeignete Werte für die Tuning-Parameter in relativ kurzer Zeit zu finden und nicht die optimale Parameterkonfiguration unabhängig von der verbrauchten Zeit zu bestimmen. Deshalb sind nach Ansicht der Autoren in [BWH13] die globalen Suchalgorithmen für das Online-Autotuning weniger gut geeignet.

Der nächste Abschnitt skizziert allgemeine Herausforderungen und Probleme, mit denen die Entwickler eines Autotuning-Ansatzes konfrontiert werden.

4.3.3. Herausforderungen und Probleme bei der Entwicklung eines Autotuning-Ansatzes

Die Entwicklung eines Autotuning-Ansatzes stellt die Entwickler vor folgende Herausforderungen:

- Moderne Hardwaresysteme haben sehr unterschiedliche Eigenschaften. Deshalb können Optimierungstechniken und Parameter, die auf einer Plattform gute Laufzeiten liefern, auf einer anderen Plattform zu sehr langsamen Laufzeiten führen.

- Bei einigen Anwendungen, wie z. B. den in dieser Arbeit betrachteten Algorithmen zur Lösung von AWPs, hängt die Programmperformance nicht nur von den Eigenschaften der Zielplattform, sondern auch von den Eingabedaten ab. In einem solchen Fall reicht das Offline-Tuning zur Installations- oder Compile-Zeit nicht aus. Da das zu lösende Problem erst zur Laufzeit bekannt ist, muss das Tuning ebenfalls zur Laufzeit stattfinden. Es ist aber durchaus möglich, Offline- mit Online-Tuning zu kombinieren. Beispielsweise kann das Autotuning problemunabhängiger Parameter in der Installationsphase mittels synthetischer Benchmarks erfolgen.

- Die zu optimierenden Parameter können sich gegenseitig beeinflussen.

- Die Anzahl der Programmvarianten wächst exponentiell mit der Anzahl zu evaluierender Parameter. Da die erschöpfende Suche bei einer sehr großen Anzahl von

Varianten zu lange dauern würde, sind Strategien zur Minimierung des Suchraumes notwendig.

- Die Zeit zum Durchführen des Autotunings soll beim Online-Autotuning nicht länger als die notwendige Zeit zum Lösen des Problems sein. Bei der Entwicklung eines Online-Autotuners muss daher zwischen der Dauer des Autotunings und dem optimalen Autotuning-Ergebnis abgewogen werden.

- Bei der Messung der Laufzeit oder der Energie können Schwankungen auftreten. Oft entstehen diese durch das Starten von Prozessen des Betriebssystems. Die Prozesse des Betriebssystems teilen sich die Hardware-Ressourcen mit der Applikation, was zu Wartezeiten und Laufzeitschwankungen führen kann. In der Literatur werden Störungen, die durch das Betriebssystem verursacht werden, als **Systemrauschen (engl. Background-Noise, OS-Jitter)** bezeichnet. Im Zusammenhang mit den Schwankungen der Messergebnisse stellt sich bei der Entwicklung eines Autotuners daher die Frage, ob und wie oft gegebenenfalls eine Messung wiederholt werden muss.

Im nächsten Kapitel werden verwandte Arbeiten aus unterschiedlichen Gebieten der Informatik vorgestellt, die sich ebenfalls mit dem automatischen Tuning von Software beschäftigen.

5. Darstellung verwandter Arbeiten und die Einordnung der eigenen Arbeit

In diesem Kapitel wird ein Überblick über verwandte Arbeiten anderer Autoren gegeben und der Beitrag der eigenen Arbeit in die aktuelle Forschungslandschaft eingeordnet.

5.1. Darstellung verwandter Arbeiten

Das Autotuning wurde erfolgreich bei vielen Anwendungen, u. a. der linearen Algebra, verschiedenen Solvern, Algorithmen für die digitale Signalverarbeitung (DSP) und MPI-Kommunikationsoperatoren, angewandt. Die ersten Autotuning-Bibliotheken waren **PHiPAC (Portable High Performance ANSI C)** [BACD97] und **ATLAS (Automatically Tuned Linear Algebra Software)** [WD97] für Operationen der linearen Algebra für dichtbesetzte Matrizen und Vektoren. Bei dem PHiPAC-Ansatz wurden mit Hilfe generischer Modelle die Richtlinien für einen portierbaren und effizienten ANSI C Code für die damals aktuellen Mikroprozessoren und Compiler festgelegt. PHiPAC enthält einen Codegenerator, der einen parametrisierten Code entsprechend dieser Richtlinien erzeugen kann. Der Autotuning-Prozess wird von einem zentralen Skript gesteuert. Dieser generiert für unterschiedliche Kombinationen von Compiler-Optionen und Programm-Parameter (z. B. Blockgröße, Aufrollfaktoren für Schleifen) den jeweiligen Programmcode, der dann auf der Zielarchitektur compiliert und ausgeführt wird. Als Ergebnis liefert das Skript diejenige Kombination von Programm-Parametern zurück, die auf dieser Plattform und mit diesem Compiler zum Programmcode mit der schnellsten Laufzeit geführt hat. Da system-spezifische Programmversionen für alle **BLAS**[1]-Operationen erzeugt werden, kann die Perfomance-Optimierung mit PHiPAC mehrere Tage dauern.

Das ATLAS-Projekt ist die Weiterentwicklung von PHiPAC. Beide Bibliotheken nutzen eine erschöpfende Suche zur Bestimmung der Programmvariante mit der besten Performance. PHiPAC und ATLAS sind Offline-Autotuner. Die Suche findet während der Compile-Zeit statt. In [YLR+05] wurde ein modellbasierter Ansatz für die Optimierung der Matrix-Multiplikation präsentiert. Die Autoren haben gezeigt, dass die Optimierung der Programm-Performance mittels eines Modells genauso effizient sein kann wie die empirische Suche.

In [EGD+05] wurde am Beispiel der ATLAS-Bibliothek eine empirische Suche mit einem modellbasierten Ansatz kombiniert. Das analytische Modell von Yotov [YLR+05] wurde dazu verwendet, den Suchraum möglicher Blockgrößen und Aufrollfaktoren für die anschließende empirische Suche einzugrenzen.

[1] BLAS (Basic Linear Algebra Subprograms) ist eine Bibliothek für Operationen der linearen Algebra.

FFTW [FJ05] steht für **Fastest Fourier Transform in the West** und ist eine Bibliothek zur Berechnung von **diskreten Fourier-Transformationen (DFTs)** mit Hilfe einer **schnellen Fourier-Transformation (engl. FFT (Fast Fourier Transform))**. Im Gegensatz zu ATLAS besteht beim FFTW der Autotuning-Suchraum nicht aus Programmparametern, sondern aus unterschiedlichen FFT-Algorithmen zur Lösung von DFTs, die Solver genannt werden. Es gibt eine Reihe unterschiedlicher FFT-Algorithmen sowie zahlreiche Varianten davon [Mer13]. Diese unterscheiden sich darin, wie bestimmte Teile einer DFT so umgeformt werden (z. B. durch Faltung), dass zur Berechnung weniger Multiplikationen (stellen den größten Rechenaufwand dar), notwendig sind. Der bekannteste FFT-Algorithmus ist der **Cooley-Tukey-Algorithmus** [CT65]. Dieser funktioniert nur für DFTs der Größe $2n$, wobei n die Anzahl der Stützstellen bezeichnet. Für eine DFT beliebiger Größe werden andere FFT-Algorithmen, wie z. B. der **Bluestein-FFT-Algorithmus** [Blu70], verwendet. Die FFTW-Bibliothek nutzt einen **Planer (engl. planner)**, um die Berechnung einer DFT auf der Zielplattform zu optimieren. Der Planer zerlegt die Berechnung einer DFT in kleine Teilprobleme, die dann rekursiv mit Hilfe unterschiedlicher FFT-Algorithmen gelöst werden. Ein Codefragment zur Lösung einer kleinen DFT wird **Codelet** genannt. Die rekursive Berechnung einer DFT wird **Plan** genannt. Dieser setzt sich aus unterschiedlichen Codelets zusammen. Die Codelets werden automatisch durch einen Compiler erzeugt. Ein Plan wird vor der eigentlichen Berechnung generiert, sobald die Größe des zu lösenden DFT-Problems bekannt ist. Zur Auswahl des schnellsten Plans (Kombination von Algorithmen) für die jeweilige Zielplattform wird dynamische Programmierung und die Zeit als Optimierungsmetrik verwendet.

Spiral [PMJ+05, PFV11] und **UHFFTW** [MJ01] sind Autotuning-Bibliotheken aus dem Bereich der Bild- und Signalverarbeitung. Spiral ist eine Bibliothek zur automatischen Optimierung von Algorithmen zur Berechnung von DSP-Transformationen. Zu diskreten DSP-Transformationen zählen u. a. DFT und diskrete Kosinustransformation. Spiral verwendet eine empirische Suche, die zur Compile-Zeit stattfindet. Der Suchraum setzt sich aus unterschiedlichen Algorithmen und Optimierungen auf der Codeebene zusammen. Der Anwender gibt die zu lösende DSP-Transformation und, wenn bekannt, zugleich die Größe der Eingabe an. Der domänenspezifische Codegenerator erzeugt daraufhin mehrere zu dieser Transformation zugehörige Algorithmen, die in der so genannten SPL-Sprache (Signal Processing Language) [XJJP01, PMJ+05] spezifiziert werden. Bei der Codegenerierung werden verschiedene Codetypen unterstützt: skalar, parallel, parametrisiert, rekursiv. Je nach Codetyp werden unterschiedliche Techniken zur Codeerzeugung verwendet. Durch die Angabe von Compiler-Direktiven können auch Optimierungen auf der Codeebene durchgeführt werden, wie z. B. die Auswahl geeigneter Aufrollfaktoren für Loop-Unrolling oder das Inlining von Konstanten. Der Spiral-Compiler generiert dann für jedes SPL-Programm den entsprechenden C-Code. Dieser wird auf der Zielarchitektur ausgeführt und deren Laufzeit wird gemessen. Von allen Codevarianten wird dann die Variante mit der besten Laufzeit gewählt. Im Rahmen des SPIRAL-Projekts wurde mit unterschiedlichen Suchstrategien experimentiert, u. a. mit der vollständigen Suche [PMS+04], dem maschinellen Lernen [dVP10], dem bandit-based

Monte-Carlo-Algorithmus [dRVP09] und der evolutionären Suche [PMS+04].

Die **UHFFT (Adaptive and Portable Software Library for Fast Fourier Transforms)**-Bibliothek arbeitet ähnlich wie FFTW. Zur Installationszeit werden automatisch optimale Faktoren für die Zerlegung einer DFT in kleine Teilprobleme bestimmt, für die in der Bibliothek effiziente DFT-Codelets existieren.

Die **OSKI (Optimized Sparse Kernel Interface)**-Bibliothek [VDY05, BLYD12] bietet auf die Laufzeit optimierte Berechnungskernels für Operationen der linearen Algebra für dünnbesetzte Matrizen. OSKI durchsucht automatisch eine Vielzahl von Programmvarianten mit unterschiedlichen Abspeicherungsformaten der dünnbesetzten Matrizen sowie unterschiedlichen Codeoptimierungen. Weil die Information darüber, wie viele Einträge der Matrix Null sind, erst zur Laufzeit verfügbar ist, kombiniert OSKI Online- mit Offline-Autotuning. In der Offline-Phase werden mehrere Codevarianten für die von der Eingabe unabhängigen zu optimierenden Parameter generiert. Die Performance dieser Varianten wird anhand kleiner Benchmarks (synthetischer Matrizen) auf der Zielarchitektur gemessen. In der Online-Phase wird dann ein heuristisches Modell zusammen mit den Ergebnissen dieser Benchmarks dazu genutzt, die Implementierung mit der besten Performance vorherzusagen. In [BLYD12] wurde eine parallele Version der OSKI Bibliothek beschrieben.

Das **SALSA (Self-Adapting System for Linear Solver Selection)**-Projekt [EF10] verwendet Bayes-Klassifikatoren, um einen geeigneten linearen bzw. nicht-linearen Solver zusammen mit einem Vorkonditionierer und weiteren Parametern zum Lösen von (nicht-)linearen GLS zu bestimmen.

Autotuning funktioniert nicht nur auf CPU-Systemen, sondern auch auf anderen Plattformen. In [DBLG11] wurde ein Autotuning Framework für FFTs auf GPUs präsentiert. Zur Reduktion des Suchraumes wurden Heuristiken verwendet. In jüngster Zeit sind auch viele Publikationen entstanden, die sich mit der automatischen Codeoptimierung auf parallelen und heterogenen Plattformen beschäftigen. **PEPPHER (Performance Portability and Programmability for Heterogeneous Many-core Architectures)** [BPT+11] ist ein Framework für die Programmierung und Optimierung von rechenintensiven Aufgaben/Tasks auf heterogenen Architekturen. PEPPHER bietet keine automatische Generierung von Implementierungsvarianten an, kann aber zum dynamischen Scheduling der Tasks [ATNW11] sowie zur Auswahl der besten Implementierungsvarianten und geeigneter Optimierungsparameter genutzt werden. Die Auswahl der besten Variante hängt von der Laufzeit und der Verfügbarkeit von Rechendaten und Hardwareressourcen ab. In [ATNW11] wurde **PEPPHER** als ein Task-Scheduler realisiert, der zum Lösen eines Scheduling-Problems unterschiedliche Heuristiken, wie z. B. die Heterogeneous Earliest Finish Time [THW02], verwendet. Der Ansatz in [DEK11] stellt eine Erweiterung von [BPT+11] dar. In [DEK11] wurde maschinelles Lernen verwendet, um in der Offline-Phase die Laufzeit von Programmvarianten anhand kleiner Testbeispiele vorherzusagen. Die daten-parallelen Programmvarianten wurden mit Hilfe algorithmischer Skelette erzeugt. Bei der Programmierung mit algorithmischen Skeletten [DEK11] stehen dem Programmierer nur eine Menge vordefinierter parametrisierter Strukturen zur Codeimplementierung zur Verfügung. Diese können jedoch in der späteren Imple-

mentierungsphase ausgebaut werden. In [DEK11] wurde eine dynamische Anpassung eines Skeletts an die Zielarchitektur implementiert. Dynamische Anpassung bedeutet im diesem Fall die automatische Auswahl geeigneter Skelettparameter. Die Auswahl der Skelettparameter erfolgt abhängig von der Größe der Eingabe mit Hilfe eines heuristischen Algorithmus aus dem Bereich der Genetischen Programmierung (GP). Der Algorithmus iteriert über den Suchraum möglicher Skelettparameter und misst die Zeit für jeden Skelettparameter und für verschiedene Eingabegrößen. Als Ergebnis liefert der GP-Algorithmus eine Abbildung der Eingabegrößen auf optimale Skelettparameter.

Active Harmony [TH11] ist ein Autotuning-System für die Optimierung von wissenschaftlichen Anwendungen auf parallelen Architekturen. Active Harmony verbindet die Idee der **Feedback-basierten Optimierung (engl. feedback-oriented optimization)** mit der **Just-in-time-Kompilierung (JIT)**. Die Feedback-basierte Optimierung ist ein allgemeiner Begriff für die Technologien, die zur Laufzeit gesammelten Informationen zur Programmoptimierung nutzen [Smi00]. Bei der Just-in-time-Kompilierung werden Programme zur Laufzeit kompiliert. Zur Definition von Optimierungsparametern und deren Abhängigkeiten untereinander wird **Constraint Specifaction Language (CSL)** verwenden. Außerdem können die CSL-Ausdrücke vom Benutzer dazu genutzt werden, Hinweise an den Suchalgorithmus zu übermitteln. Mögliche Hinweise können zum Beispiel die Defaultwerte für Optimierungsparameter, die Bedingungen an die Größe von MPI-Nachrichten oder die Anzahl zu startender MPI-Threads sein. Active Harmony nutzt einen parallelen Suchalgorithmus, genannt **Parallel Rank Order (PRO)**. Dieser basiert auf der Simplex-Suche. Im Gegensatz zu anderen Suchalgorithmen, erlaubt der PRO-Algorithmus das gleichzeitige Auswerten unterschiedlicher Codevarianten und Parameter. Im Fall von SPMD-Programmen können unterschiedliche Codevarianten und Parameter simultan auf unterschiedlichen CPU-Knoten eines Rechensystems ausgewertet werden. Das Verwenden eines parallelen Suchalgorithmus reduziert die Anzahl notwendiger Iterationen zum Finden einer nahezu optimalen Programmvariante. Das Compilieren von neuen Codevarianten findet auf den für diese Aufgabe speziell reservierten Rechenknoten statt, so dass die Auswertung von Programmvarianten auf anderen Knoten nicht unterbrochen werden muss.

Perpetuum [KP11a, KP11b] ist ein Autotuner für parallele Programme, der in das Betriebssystem Linux integriert ist. Perpetuum ist ein Online-Autotuner zur Laufzeitoptimierung von parallelen Implementierungen mit gemeinsamem Adressraum. Als Suchmethode nutzt Perpetuum eine Variante des Simplex-Algorithmus. Die zu optimierenden Anwendungen müssen iterativ sein. Das heißt, der Programmcode soll einen Bereich enthalten, der ständig wiederholt wird. Solche Bereiche eines Programmcodes werden als "hot spots" bezeichnet. Ein Beispiel dafür sind berechnungsintensive Schleifen. Vor Beginn des Autotunings muss der Anwender dem Betriebssystem mitteilen, welche "hot spots" im Programm existieren und welche Parameter optimiert werden sollen. Ein Nachteil von Perpetuum ist es, dass keine Generierung von neuen Codevarianten unterstützt wird. Deshalb können nur die Optimierungstechniken evaluiert werden, die keine Coderestrukturierung erfordern. Im Gegensatz zu anderen Autotunern ist Perpetuum in der Lage, mehrere parallele Anwendungen gleichzeitig zu optimieren.

PLASMA (Parallel Linear Algebra Software for Multi-core Architectures) und **MAGMA (Matrix Algebra on GPU and Multi-core Architectures)** [ADD+09] sind zwei Projekte, die sich mit der Optimierung von Algorithmen der linearen Algebra auf Mehrkernprozessoren beschäftigen. PLASMA ist eine Neuimplementierung der Bibliotheken LAPACK [ABD+90] und ScaLAPACK [CDD+96] für Mehrkernprozessoren mit gemeinsamem Speicher. Das MAGMA-Projekt hat die Entwicklung einer Bibliothek der linearen Algebra für dünnbesetzte Matrizen zum Ziel, ähnlich der LAPACK-Bibliothek, aber für heterogene Architekturen (bestehend aus CPUs und GPUs). Bei MAGMA werden Algorithmen der linearen Algebra in Tasks unterschiedlicher Granularität partitioniert. Kleine, nicht parallelisierbare Tasks werden auf CPUs ausgeführt, während große Tasks, wie z. B. die BLAS3-Routinen, auf GPUs ausgeführt werden [DDG+12]. Die MAGMA-Bibliothek enthält einen Autotuner für die Matrix-Multiplikationsroutine GEMM auf GPUs [LDT09]. In [STD12] wurden die Loop-Tiling-Algorithmen für die LU, QR und Cholesky-Zerlegungen [BLKD09] an heterogene CPU/GPU-Architekturen angepasst. Die heterogenen Loop-Tiling-Algorithmen schlagen zwei Arten von Blockgrößen vor: Für CPU- und GPU-Kerne. In [STD12] werden die optimalen Blockgrößen für die CPU- und GPU-Kerne automatisch ausgewählt. Im Rahmen der PLASMA-Bibliothek wurde in [ADNT11] ein Autotuner für die QR-Zerlegung entwickelt.

Autotuning-Techniken wurden auch erfolgreich zur Optimierung von Stencilberechnungen (Nachbarschaftsberechnungen) auf CPUs und GPUs eingesetzt. Bei einer Stencilberechnung wird in jedem Zeitschritt jedem Punkt sowie benachbarten Punkten des Gitters entsprechend einer Berechnungsfunktion ein neuer Wert zugeordnet. Stencilberechnungen finden ihre Anwendung in vielen Algorithmen, einschließlich den Algorithmen zur Lösung von PDEs. Ein allgemeiner Framework zur automatischen Performance-Optimierung von Stencilberechnungen wurde von Kamil und anderen in [KCW+10] präsentiert. Die Beschreibung eines Stencils wird geparst und als ein **abstrakter Syntaxbaum** (engl. Abstrakt Syntax Tree (AST)) dargestellt. Danach werden auf dem AST unterschiedliche Optimierungen und Transformationen, wie z. B. das Loop-Unrolling und Loop-Tiling, ausgeführt. Der parallele Codegenerator verwendet POSIX-Threads zur Generierung parallelen Codes für Rechner mit gemeinsamem Adressraum. Der parallele Code wird dann mit unterschiedlichen Parameterkombinationen auf der Zielarchitektur ausgeführt und für jede Parameterkombination wird die Laufzeit gemessen. Am Ende des Autotuning-Prozesses wird dem Benutzer die schnellste Parameterkombination mitgeteilt. In [KCO+10] wurde der Autotuning-Ansatz von [KCW+10] für die parallele Programmierung mit CUDA auf GPUs erweitert.

PATUS (Parallel Auto-Tuned Stencils)[CSB11, Chr11b] ist ebenfalls ein Autotuner für Stencilberechnungen auf Mehrkernprozessoren (bestehend aus CPUs und GPUs), der aber aktuell nur die Traversierung des Gitters mit Hilfe der Jacobi-Iteration unterstützt [Chr11b]. Bei der Jacobi-Iteration dürfen keine Datenabhängigkeiten zwischen den Gitterpunkten bestehen. Bei der Anwendung von Stencilberechnungen auf Gitterpunkte spielt die Reihenfolge der Berechnungen keine Rolle. PATUS bietet mehrere Strategien zur Traversierung des Suchraums und Parallelisierung des Sourcecodes. Die von PA-

TUS unterstützen Optimierungen sind u. a. die Vektorisierung, Loop-Unrolling, Loop-Blocking und die spezielle Optimierungen für NUMA-Architekturen, wie z. B. Thread-Affinity [HW10] und Cache-Bypassing [Dat09, Chr11b]. Vom Codegenerator erzeugte Varianten werden auf der Zielarchitektur ausgeführt. Als Ergebnis des Autotunings liefert PATUS die beste Optimierungs- und Parallelisierungsstrategie.

Autotuning-Techniken wurden auch zur Performance-Verbesserung von MPI-Kommunikationsoperationen eingesetzt. **STAR-MPI (Self Tuned Adaptive Routines for MPI Collective Operations)** [FYL06] ist ein Autotuner zur automatischen Optimierung von kollektiven MPI-Kommunikationsoperationen. Für jede kollektive MPI-Kommunikationsoperation hält STAR-MPI eine Menge von Kommunikationsalgorithmen bereit. Welcher Kommunikationsalgorithmus für die jeweilige MPI-Kommunikationsoperation der effizienteste ist, hängt vor allem von der Anwendung und den Eigenschaften der Zielplattform ab. Um die Anzahl zu testender Kommunikationsalgorithmen klein zu halten, werden die Kommunikationsalgorithmen entsprechend den Optimierungszielen und MPI-Operationen in Gruppen unterteilt. Zum Beispiel werden solche Kommunikationsalgorithmen in eine Gruppe zusammengefasst, die Kommunikationsoperationen für kleine Nachrichtengrößen optimieren. Das Autotuning findet zur Laufzeit statt. Zuerst wird aus jeder Gruppe ein Algorithmus ausgewählt und die Laufzeiten dieser Algorithmen werden untereinander verglichen. Steht die Gewinner-Gruppe fest, so werden alle Algorithmen aus dieser Gruppe untereinander verglichen und der schnellste Algorithmus bestimmt.

Moderne Compiler bieten eine Reihe von Techniken zur Codeoptimierung. Der Benutzer kann durch die Angabe der Compiler-Flags bestimmte Techniken der Codeoptimierung an- und ausschalten. Mit der Aktivierung des „-Ox“ Flags kann der Optimierungslevel x gewählt werden. Je höher der Optimierungslevel, desto mehr Optimierungen werden bei der Kompilierung eines Programms verwendet. Jedem Optimierungslevel „-O1“ bis „-O3“ ist eine feste Menge von Optimierungen, basierend auf der Erfahrung von Compilerentwicklern, zugeordnet. Generell ist die beste Kombination von Compiler-Optimierungen architektur- und problemabhängig. Außerdem können sich verschiedene Compiler-Optimierungen untereinander beeinflussen. Viele Arbeiten aus dem Compilerbereich befassen sich mit der automatischen Auswahl der besten Kombination von Compiler-Optimierungen (Compiler-Flags). **PEAK** [PE08] ist ein Autotuning, welches für die performance-relevanten Teile eines Programms die beste Kombination der Compiler-Optimierungen ermittelt. In [PE08] wurde der Algorithmus **Combined Elimination (CE)** vorgeschlagen. Der CE-Algorithmus entfernt iterativ aus der Menge zulässiger Optimierungen solche Optimierungen, die nicht zur Performance-Verbesserung beitragen. Der Algorithmus der **statistischen Auswahl (engl. Static Selection)** in [PKHW04] nutzt orthogonale Felder, um die Auswirkung unterschiedlicher Compiler-Optimierungen auf die Performance abzuschätzen. Die Autoren in [CFA+07] und [FKM+11] nutzten die Methoden des maschinellen Lernens, um die Anzahl zu evaluierender Compiler-Optimierungen einzuschränken. Zur Vorhersage der besten Kombination von Compiler-Optimierungen nutzt das Modell in [CFA+07] neben den Laufzeiten vorheriger Programmläufe auch Perfomance-Counter.

Cetus [BML+13] ist eine Compiler-Infrastruktur zur automatischen Parallelisierung von Programmen. Der Cetus-Compiler verwendet eine Offline-Autotuning-Strategie, die als Fenster-basiertes Autotuning bezeichnet wird. Bei dieser wird das Programm in Teile, auch p-Fenster genannt, aufgeteilt. Die Größe eines p-Fensters wird durch die Anzahl der darin enthaltenen Schleifen bestimmt und vom Benutzer vorgegeben. Die Optimierungsparameter werden mittels eines Abhängigkeitsgraphen in Gruppen, als o-Fenster bezeichnet, aufgeteilt. Die voneinander unabhängigen Optimierungsparameter gehören unterschiedlichen o-Fenstern an und können deshalb gleichzeitig evaluiert werden. Zu jedem p-Fenster wird anhand von Laufzeiten die beste Kombination der Parameter in jedem der o-Fenster bestimmt. Bei der Optimierung werden die Abhängigkeiten zwischen den aneinander angrenzenden Schleifen der p-Fenster berücksichtigt. Zur Traversierung des Suchraumes wird eine erschöpfende Suche eingesetzt.

In [KGCF13] wurde der Insieme-Compiler verwendet, um die Tasks dynamisch, basierend auf einem Vorhersagemodell, auf vorhandene OpenCL-fähige Geräte zu verteilen. Bei der Erzeugung eines Vorhersagemodells wurde mit unterschiedlichen Methoden des maschinellen Lernens experimentiert.

In [JTD+12] wurde ein mehrkriterielles Autotuning-Framework für parallele Programme vorgestellt. Das Framework besteht aus einem Compiler und einem Laufzeitsystem. Der mehrkriterielle Autotuner erzeugt eine ausführbare Datei mit mehreren Programmversionen, die eine Menge von Kompromisslösungen zwischen verschiedenen Optimierungszielen repräsentieren. Als Beispiel für die Anwendbarkeit des Verfahrens wurde die parallele Berechnung der Matrixmultiplikation betrachtet. Die Blockgröße und die Anzahl der Threads soll so gewählt werden, dass die Ausführungszeit minimiert und die Effizienz maximiert wird.

Die Methode der Programmspezialisierung optimiert das Programm hinsichtlich der Programminvarianten unter der Berücksichtigung von Eingabedaten. In [OKJ+13] wurde eine Methode zur automatischen Schleifenspezialisierung für sequentielle Programme vorgestellt. Die von den Autoren vorgeschlagene Technik wird „Invariant-induced Pattern based Loop Specialization (IPLS)“ genannt. Diese Technik kann Muster erkennen, die sich über Iterationen von Schleifen hinweg für bestimmte Eingaben wiederholen. ILPS spezialisiert die Schleife durch das Entrollen derselben mit der Länge des Musters. Jede Iteration der entrollten Schleife wird dann auf den entsprechenden Wert des Musters gesetzt. Der Prototyp von ILPS wurde im LLVM-Compiler-Framework [LA04] implementiert.

Die Tabelle 5.1 fasst die unterschiedlichen Arbeiten aus dem Autotuning Bereich zusammen.

5.2. Darstellung der eigenen Arbeit

Den Kernpunkt dieser Arbeit stellt die Entwicklung eines selbstadaptiven Algorithmus zum effizienten numerischen Lösen von Anfangswertproblemen gewöhnlicher Differentialgleichungssysteme auf modernen parallelen Architekturen dar.

Generell hängt die Effizienz eines parallelen Programms stark von den Eigenschaften

Tabelle 5.1.: Übersicht ausgewählter Arbeiten im Autotuning-Bereich.

Projekt	Publikationen	Parallelität	Domäne
PHiPaC	[BACD97]	nein	dichtbesetzte Matrizen
ATLAS	[WD97, EGD+05, YLR+05]	nein	dichtbesetzte Matrizen
FFTW	[FJ05]	nein	DFT
FFTW	[DBLG11]	ja	DFT
Spiral	[PMS+04]	nein	DSP-Transformationen
UFFTW	[MJ01]	nein	DSP-Transformationen
OSKI	[VDY05]	nein	dünnbesetzte Matrizen
OSKI	[BLYD12]	ja	dünnbesetzte Matrizen
SALSA	[EF10]	nein	(nicht-)lineare GLS
PLASMA	[ADNT11]	ja	dichtbesetze Matrizen
MAGMA	[LDT09, STD12]	ja	dichtbesetze Matrizen
PEPPHER	[BPT+11]	ja	allgemein
Active Harmony	[TH11]	ja	allgemein
Perpetuum	[KP11a, KP11b]	ja	allgemein
STAR-MPI	[FYL06]	ja	MPI-Operationen
Kamils Autotuner	[KCW+10, KCO+10]	ja	Stencilberechnungen
PEAK	[PE08]	nein	allgemein
Patus	[CSB11, Chr11b]	ja	Stencilberechnungen
Cetus	[BML+13]	ja	allgemein
Aufteilung der Tasks basierend auf Insieme-Compiler	[KGCF13]	ja	Aufteilung der Tasks auf heterogenen Architekturen
IPLS	[OKJ+13]	nein	Programmspezialisierung

der Zielplattform ab, wie z. B. der Multicore-Architektur, der Cachearchitektur sowie der Topologie und der Bandbreite des Speichers. Um eine gute Performance und hohe Skalierbarkeit erreichen zu können, muss der (parallele) Programmcode für die Zielplattform optimiert werden. Da aber die neuen Prozessorgenerationen in immer kürzeren Abständen auf den Markt kommen und die Komplexität und Heterogenität der Hardware immer weiter zunimmt (vgl. Kapitel 4.1), wird das manuelle Optimieren des Programmcodes stets schwieriger und zeitaufwendiger. Das Problem des manuellen Optimierens lässt sich jedoch durch den Entwurf selbstadaptiver Techniken und Algorithmen umgehen.

Numerische Verfahren zur Lösung von Anfangswertproblemen gewöhnlicher Differentialgleichungssysteme approximieren die Lösungsfunktion durch die Berechnung einer Folge von Zeitschritten, solche Lösungsverfahren werden allgemein als **Zeitschrittverfahren (engl. time stepping methods)** bezeichnet (vgl. Kapitel 3). Die Lösung eines AWP kann sehr rechenintensiv sein, d. h. sehr hohen Zeitaufwand und viele Zeitschritte

benötigen. Abhängig von dem gestellten AWP, der vorgegebenen Genauigkeitstoleranz und der Länge des Integrationsintervalls, kann die Anzahl der zur Lösung benötigten Schritte im Bereich von mehreren Tausenden bis Millionen liegen. Damit die AWPs zukünftig schneller und möglicherweise energieeffizienter gelöst werden können, ist eine effiziente Umsetzung von ODE-Verfahren auf modernen Architekturen erforderlich. Da die Laufzeit der ODE-Verfahren von den Eigenschaften der Hardware und dem zu lösenden ODE-Problem abhängig ist, muss sich der Programmcode an diese Eigenschaften automatisch anpassen können.

In dieser Arbeit wird ein Autotuning-Algorithmus präsentiert, der eine automatische Anpassung von ODE-Verfahren an spezifische Eigenschaften der Zielplattform und des ODE-Systems erlaubt. Der Autotuning-Algorithmus nutzt die zeitschrittorientierte Berechnungsstruktur der Lösungsverfahren aus, um zur Laufzeit die schnellste Implementierungsvariante aus der Menge verfügbarer Implementierungen auszuwählen. Die Anwendbarkeit des Autotuning-Ansatzes auf ODE-Verfahren zur Lösung von AWPs wird anhand einer Klasse expliziter PC-Verfahren vom Runge-Kutta-Typ gezeigt. Diese Verfahren wurden aufgrund ihres großen Parallelisierungspotenzials bezüglich des Systems und der Methode gewählt. Die spezielle Berechnungsstruktur expliziter PC-Verfahren vom Runge-Kutta-Typ erlaubt die Erzeugung einer Vielzahl von Implementierungsvarianten, die sich in den Datenstrukturen und der Reihenfolge der Schleifen innerhalb eines Zeitschrittes unterscheiden und somit unterschiedlich in ihren Lokalitäts- und Skalierbarkeitsverhalten sind. Die Menge der erzeugten Implementierungsvarianten wird als Kandidatenpool bezeichnet. Der Autotuning-Algorithmus klassifiziert die zu lösenden ODE-Systeme anhand deren Klassenzugehörigkeit in Gruppen. Zu jeder Gruppe stellt der Algorithmus eine Menge von Implementierungsvarianten aus dem Kandidatenpool bereit. Der Vergleich von Implementierungsvarianten findet zur Laufzeit statt, während der ersten Zeitschritte des ODE-Verfahrens. Die beste Implementierungsvariante wird anhand eines empirischen Vergleichs der Laufzeiten gewählt.

Im Kandidatenpool sind unter anderem Implementierungsvarianten enthalten, die durch Anwendung von Loop-Tiling entstanden sind und für die eine geeignete Blockgröße gewählt werden muss. Bei Loop-Tiling-Varianten hat die Wahl der Blockgröße einen entscheidenden Einfluss auf die Laufzeit des Programmcodes. Die Bestimmung einer guten Blockgröße ist eine schwierige Angelegenheit. Wird die Blockgröße zu klein gewählt, dann kann der Cache nicht voll ausgenutzt werden und der Overhead der Schleifenkontrolle kann den Laufzeitgewinn des Loop-Tilings zunichte machen. Wird die Blockgröße zu groß gewählt, passen unter Umständen nicht alle Daten eines Blocks in den Cache hinein. Dies kann zu einer Vergrößerung der Anzahl von Konfliktfehlzugriffen führen. Die Bestimmung einer geeigneten Blockgröße für die im Kandidatenpool enthaltenen Loop-Tiling-Varianten ist ein wichtiger Teil dieser Arbeit. Zur Traversierung des Suchraumes potenzieller Blockgrößen wird eine geführte Suche verwendet. Als erster Schritt wird der Suchraum mit Hilfe eines analytischen Modells eingeschränkt. Dadurch entsteht eine weitaus kleinere Menge von Blockgrößen, die dann zur Laufzeit empirisch evaluiert werden können. Das analytische Modell basiert auf der Berechnung von **Arbeitsräumen (engl. working spaces)** der in den Impelementierungsvarianten auftretenden Schleifen.

Der Arbeitsraum einer Schleife ist die Menge aller darin referenzierten Datenelemente.

Um den Autotuning-Algorithmus für parallele Varianten des PC-Verfahrens vom RK-Typ zu entwickeln, wurde in dieser Arbeit wie folgt vorgegangen:

- Zunächst erfolgte eine Untersuchung der Anwendbarkeit des Autotuning-Ansatzes auf sequentiellen Varianten des betrachteten Verfahrens. Daraus entstand ein Grundalgorithmus des Autotunings, der vorerst nur in der Auswahl einer guten Implementierungsvariante bestand und die Programmparameter, wie z. B. Blockgröße, unberücksichtigt ließ [KKR10]. Im nächsten Schritt wurde dieser um die automatische Auswahl von Blockgrößen ergänzt [KKR11a, KKR11b].

- Danach erfolgte die Übertragung der sequentiellen Autotuning-Konzepte auf parallele Implementierungsvarianten mit gemeinsamem Adressraum [KKR14]. Zur automatischen Auswahl von Blockgrößen wurde, wie schon bei sequentieller Ausführung, eine modellgeführte Suche verwendet. Für parallele Loop-Tiling-Varianten ist das Modell der Blockgrößenauswahl weiterentwickelt worden. Das resultierende Modell berücksichtigt die komplexe Speicherhierarchie der Multicore-Systeme, insbesondere die Gegebenheit, dass sich mehrere Kerne einen Cache-Speicher teilen können. Beim weiterentwickelten Modell werden die Blockgrößen für alle Arbeitsräume und alle Cachestufen vorselektiert. Die beste Blockgröße wird mit Hilfe einer heuristischen empirischen Suche aus der vorselektierten Menge der Blockgrößen gewählt.

  Wie bereits erwähnt, findet der empirische Vergleich von Blockgrößen und Implementierungsvarianten zur Laufzeit statt. Deshalb ist es besonders wichtig, nicht nur die Anzahl zu evaluierender Blockgrößen, sondern auch der Implementierungsvarianten so weit wie möglich einzuschränken. Während der Lösung eines Anfangswertproblems sollten möglichst nur die Implementierungsvarianten evaluiert werden, die auf der Zielarchitektur eine hohe Effizienz erreichen können. Der entwickelte Autotuning-Algorithmus enthält eine Strategie zur Reduktion der Anzahl zu evaluierender Implementierungsvarianten. Die Strategie basiert auf der Abschätzung der Zeit, die zur Prozesssynchronisation benötigt wird. Aus dem Kandidatenpool werden die Implementierungsvarianten aussortiert, die im Vergleich zu anderen einen viel größeren Anteil der Laufzeit zur Prozesssynchronisation benötigen [KKR14].

- Im Rahmen der Masterarbeit von Frank Wein [Wei14] wurde ein selbstadaptiver ODE-Solver für verteilten Adressraum entwickelt. Dieser unterstützt verschiedene Optimierungsmöglichkeiten: die beste Implementierungsvariante kann bezüglich der Laufzeit, des Energieverbrauchs oder des Energy-Delay-Produkts gewählt werden. Zusätzlich bietet der Autotuner die Möglichkeit der automatischen Anpassung der Anzahl parallel rechnender Prozesse zur Minimierung des Energieverbrauchs bzw. der zur Lösung benötigten Laufzeit.

5.3. Wissenschaftliche Einordnung der Arbeit

Für viele numerische Grundroutinen, wie z. B. Matrixmultiplikation [WPD01, BACD97] oder FFT [FJ05], sind in den letzten Jahren hochspezialisierte Autotuning-Bibliotheken entwickelt worden (s. Abschnitt 5.1). Im Bereich der ODE-Verfahren existiert bisher keine solche Bibliothek, die für das zu lösende AWP und die verwendete Rechnerarchitektur automatisch eine auf Performance optimierte Implementierungsvariante liefert.

In dieser Arbeit sind Autotuning-Techniken speziell für ODE-Verfahren entworfen worden. Im Gegensatz zu Algorithmen der linearen Algebra, wie z. B. der Matrixmultiplikation, weisen die ODE-Verfahren eine viel größere Abhängigkeit der Laufzeit von der Eingabe auf. Die entwickelten Autotuning-Algorithmen nutzen daher die charakteristischen Eigenschaften des zu lösenden ODE-Systems, wie die Systemgröße und das Zugriffsmuster der Funktionsauswertung, zur Bestimmung einer effizienten Implementierungsvariante und geeigneter Programmparameter.

Die meisten in Abschnitt 5.1 beschriebenen Ansätze sind entweder eingeschränkt auf eine spezielle Domäne oder unterstützen kein Autotuning für parallele Programme.

Die allgemeinen Autotuning-Ansätze zur Performance-Optimierung paralleler Programme wie Active Harmony [TH11], Perpetuum [KP11a, KP11b] oder Cetus [BML+13] sind rein empirische Ansätze, die im Vergleich zu den in dieser Arbeit entwickelten Autotuning-Algorithmen kein Modell zur Reduktion des Suchraumes von Blockgrößen nutzen.

6. Techniken zur Steigerung der Lokalität von Programmen und Codeoptimierung mittels Vektorisierung

In diesem Kapitel werden zunächst der prinzipielle Aufbau und die Funktionsweise eines Caches beschrieben. Darauf aufbauend werden die Prinzipien der zeitlichen und der räumlichen Lokalität von Programmen erläutert. Es folgt die Darstellung der Techniken zur Optimierungen der Speicherzugriffslokalität und die Beschreibung der Programmoptimierung mittels Vektorisierung. Am Ende des Kapitels wird auf Probleme der Codeoptimierung durch Compiler eingegangen.

6.1. Funktionsweise von Caches und das Prinzip der Lokalität

Die Laufzeit eines Programms hängt nicht nur von den Eigenschaften der auszuführenden Zielplattform, sondern auch wesentlich von der Reihenfolge der Berechnungen und von der Reihenfolge der Zugriffe auf die Daten im Speicher ab. Der Grund dafür ist die **Lokalitätseigenschaft der Speicherzugriffe (engl. locality of reference)** von Programmen. Unter der Lokalitätseigenschaft der Speicherzugriffe versteht man die empirische Beobachtung, dass die meisten Programme zu einem beliebigen Zeitpunkt nur auf einen kleinen Teil des Gesamtadressraumes zugreifen. Die grobe Abschätzung ist, dass ein Programm zu 90 % der Laufzeit nur 10 % des gesamten Programmcodes ausführt [Chi01]). Das ist der Fall, weil die meisten Programme aus Schleifen bestehen und deshalb sich Programmteile oft wiederholen. Man unterscheidet zwei Arten von Lokalität [PH13]:

- Die **räumliche Lokalität** besagt: Wenn auf eine Speicherzelle zugegriffen wird, so wird auch bald mit hoher Wahrscheinlichkeit auf die benachbarten Speicherzellen zugegriffen.
- **Zeitliche Lokalität** bedeutet: Nach einem Zugriff auf eine Speicherzelle wird mit hoher Wahrscheinlichkeit zu einem bald folgenden Zeitpunkt erneut auf diese Speicherzelle zugegriffen.

Die Lokalitätseigenschaft der Speicherzugriffe bildet die Grundlage der hierarchischen Speicherarchitektur moderner Rechnersysteme. Je höher in der Speicherhierarchie, desto schneller, kleiner, aber auch teurer (gemessen in Kosten pro Bit) ist der Speicher [PH13]. Die oberste Ebene der Hierarchie bilden die CPU-Register. Diese sind schnell getaktete Speicher innerhalb des Prozessors. Der Zugriff auf CPU-Register erfolgt ohne zeitliche Verzögerung. Die unterste Ebene der Hierarchie bilden der Hauptspeicher

und die Festplatte. Zwischen der CPU und dem Hauptspeicher befinden sich meistens mehrere Caches unterschiedlicher Kapazität und Zugriffszeit. Ein Cache enthält Kopien von Daten aus dem Hauptspeicher, auf die häufig zugegriffen wird. Die Daten im Cache werden in Form von **Blöcken, auch Cachezeilen genannt (engl. cache line, cache block)**, bestehend aus mehreren zusammenhängenden Maschinenwörtern, abgelegt. Dagegen können die einzelnen CPU-Register maximal ein Wort speichern. Werden die Daten von der CPU durch Spezifikation eine Speicheradresse angefordert, so prüft der Cache-Controller, ob diese im Cache vorliegen. Falls ja, man spricht dabei von einem **Cache-Treffer (engl. cache hit)**, dann werden die Daten aus dem Cache gelesen und an den Prozessor übergeben. Befinden sich die angefragten Daten nicht im Cache, man spricht von einem **Cache-Fehlzugriff (engl. cache miss)**, dann muss zunächst der Cacheblock, der die angefragte Speicherzelle enthält, aus dem Hauptspeicher in den Cache geladen werden. Da die Caches blockbasiert organisiert sind, wird bei einem Fehlzugriff nicht nur das von der CPU angeforderte Speicherwort, sondern ein ganzer Cacheblock, der dieses Speicherwort enthält, aus dem Hauptspeicher in den Cache geladen. Die Speicherkapazität des Caches ist geringer als die des Hauptspeichers, deshalb müssen die Speicherblöcke des Hauptspeichers auf die Blöcke des Caches abgebildet werden. Die Speicherblöcke aus dem Hauptspeicher werden im Cache in **Blockrahmen (engl. block frame)** abgelegt. Jeder Blockrahmen speichert außer einem Cacheblock noch ein **Adressetikett (engl. cache tag)**, sowie einige Statusbits. Die Anzahl der Speicherplätze in einem Blockrahmen wird **Blocklänge (engl. line size)** genannt. Typische Blocklängen sind 32, 64 oder 128 Bytes. Einzelne Blockrahmen bilden zusammen einen **Satz (engl. set)**. Die Anzahl der Blockrahmen k in einem Satz wird als **Assoziativität** des Caches bezeichnet. Ist m die Anzahl der Cacheblöcke und k die Assoziativität des Caches, so lässt sich die Anzahl der Sätze in einem Cache als $n = m/k$ berechnen. Zur Organisationsform eines Caches gibt es unterschiedliche Möglichkeiten; diese bestimmen, wie die Blöcke aus dem Hauptspeicher im Cache abgelegt werden [BU10, PH13]:

- Ist $k = 1$, so spricht man von einem **direkt abbildenden** Cache. Bei dieser Form des Cache-Speichers enthält jeder Satz nur einen einzigen Blockrahmen. Der Speicherblock kann deshalb nur an eine bestimmte Position im Cache gespeichert werden.

- Ist $k = m$, so besteht der Cache-Speicher nur aus einem einzigen Satz, es handelt sich um einen **voll-assoziativen** Cache. Der Speicherblock kann an jede beliebige Position des Caches gespeichert werden.

- Bei einem k-fach **mengen-assoziativen Cache** hat man beim Abbilden eines Blocks aus dem Hauptspeicher auf den Cache k verschiedene Möglichkeiten. Ein Block B_j aus den Hauptspeicher kann im Cache in jedem Satz i mit $i = j \, mod \, k$ abgelegt werden.

Die bei der Ausführung eines Programms auftretenden Cache-Fehlzugriffe können unterschiedliche Ursachen haben [BU10, PH11] :

- Ein **Capacity Miss** tritt auf, wenn die Kapazität des Caches für die Abspeicherung der von einem Programm verwendeten Datenstrukturen (des Programmarbeitsraumes) nicht ausreicht. In Folge dessen werden die Daten, die sich im Cache befinden, aus dem Cache verdrängt.

- Ein **Conflict Miss (Konfliktfehlzugriff) oder Interferenz Miss** kann bei einem direkt-abbildenden oder bei einem mengen-assoziativen Cache auftreten. Konfliktfehlzugriffe treten auf, wenn viele Cache-Blöcke auf denselben Satz abgebildet werden. Als eine **Interferenz** bezeichnet man das Verdrängen von Daten im Cache durch den Zugriff auf andere Daten. Werden die Elemente eines Arrays im Cache durch andere Elemente des gleichen Arrays verdrängt, so spricht man von Konfliktfehlzugriffen, die in Folge von **Selbstinterferenzen (engl. self-interferences)** entstanden sind. Werden die Elemente eines Arrays im Cache durch die Elemente eines anderen Arrays verdrängt, so spricht man von **Fremdinterferenz (engl. cross-interference)**.

- Ein **Compulsory Miss** oder **Cold Miss** tritt beim allerersten Zugriff auf ein Datum auf. Die Daten befinden sich noch nicht im Cache und müssen zuerst aus dem Hauptspeicher in den Cache geladen werden. Diese Art von Cache-Fehlzugriffen ist beim ersten Zugriff auf ein Datum unvermeidbar.

Bisher wurde die Arbeitsweise eines Caches beschrieben, der die Daten des auszuführenden Programms enthält. Man spricht dabei von einem **Datencache** erster Stufe. Moderne Prozessoren, wie z. B. die Intel Haswell-Prozessoren, verfügen über mehrere Caches, die zusammen eine **Cache-Hierarchie** bilden. Die einzelnen Caches werden nach ihrer Zugriffszeit in **Cache-Stufen (engl. cache levels)** eingeteilt. Die meisten Prozessoren besitzen bis zu drei Cache-Stufen: L1, L2, L3. Der L1-Cache ist der schnellste und meistens auch der kleinste der Cache-Speicher. Werden Daten von der CPU angefragt, so wird zuerst immer der L1-Cache durchsucht. Sind die Daten im L1-Cache nicht vorhanden, so wird als nächstes auf die nächsthöhere Cachestufe bzw. den Hauptspeicher zugegriffen.

Neben den Daten des auszuführenden Programms müssen auch die Instruktionen gespeichert werden. Da in Schleifen mehrmals auf die gleichen Instruktionen zugegriffen wird, sollen auch die häufig verwendeten Instruktionen im Cache gespeichert werden. Die meisten aktuellen Prozessoren speichern die Daten des auszuführenden Programms getrennt von den Instruktionen ab. Zu diesem Zweck besitzen die Prozessoren getrennte L1-Caches. Die Instruktionen eines Programms werden im **L1-Instruktionscache** gespeichert. Die Caches L2 und L3 sind bei den meisten Prozessoren einheitlich (engl. unified caches) (Ausnahme z. B. Itanium 2 Montecito), d. h. sie können Instruktionen und Daten enthalten. Bei modernen Prozessoren sind der Daten- und der Instruktionscache durch separate Busse mit der CPU verbunden. Dadurch können Befehle und Operanden unabhängig voneinander geladen werden, was zu einer Steigerung der Rechenleistung führt.

Will man effiziente Programme schreiben, so muss man die hierarchische Speicherorganisation von Rechnersystemen berücksichtigen. Damit dies gelingt, muss der Pro-

grammcode die Cache-Lokalität bestmöglich ausnutzen. Es existieren unterschiedliche Optimierungstechniken, die darauf abzielen, räumliche bzw. zeitliche Lokalität von Programmen zu verbessern. Diese werden im nächsten Kapitel beschrieben.

6.2. Techniken zur Steigerung der Lokalität von Programmen

Eine Verbesserung der räumlichen bzw. der zeitlichen Lokalität von Programmen lässt sich durch das Ändern des Datenlayouts oder durch das Ändern der Reihenfolge von Speicherzugriffen erreichen.

6.2.1. Ändern des Datenlayouts zur Optimierung der Lokalität

Konfliktfehlzugriffe treten bei einem direkt-abbildenden oder einem mengen-assoziativen Cache auf, wenn innerhalb einer Schleife eine Instruktion mehrmals auf Daten zugreift, deren Adressen auf die gleiche Cachezeile abgebildet werden. Infolge eines erneuten Zugriffs können die Daten, die auf die gleiche Cachezeile verweisen, aus dem Cache verdrängt werden, bevor diese wiederverwendet werden können. Um Konfliktfehlzugriffe zu vermeiden, kann es hilfreich sein, die Adressen der Daten, auf die eine Instruktion zugreift, so zu ändern, dass diese auf unterschiedliche Cachezeilen abgebildet werden. Dadurch steigt die Lokalität der Datenzugriffe. Zur Optimierung der Lokalität kann das Datenlayout durch die folgenden Techniken verändert werden [MSS03, KW02, Wol95, AK02]:

- Unter der Technik des **Array-Paddings** versteht man die Allokation eines zusätzlichen Speicherplatzes (engl. **pads**), der aber im Laufe der Berechnung nicht genutzt wird [RT97]. Beim Array-Padding ändert sich entweder die Größe eines Arrays, in dem zusätzliche Zeilen oder Spalten zwischen den aufeinander folgenden Elementen eingefügt werden, man spricht dabei von einem **Intra-Array-Padding**, oder es werden Dummy-Elemente zwischen zwei aufeinander folgenden Arrays eingefügt, man spricht dann von einem **Inter-Array-Padding**. Zur Vermeidung von Konfliktfehlzugriffen kann ein Array auch an bestimmten Speichergrenzen ausgerichtet werden. Beim Array-Padding vergrößert sich einerseits der Speicherbedarf durch die eingefügten Elemente, anderseits werden die Adressen der Elemente eines Arrays verschoben, wodurch bei geeignet gewählter Größe des Pads Konfliktfehlzugriffe vermieden werden können.

- Bei einem **Array-Transpose** werden die Dimensionen eines mehrdimensionalen Arrays vertauscht. Konfliktfehlzugriffe können durch eine Verschiebung der Adressen der inneren Variablen eines Arrays vermieden werden.

- Unter **Array-Merging** versteht man das Verschmelzen von zwei Arrays gleicher Dimension und des gleichen Datentyps zu einem Array. Array-Merging dient der Verbesserung der räumlichen Lokalität. Darüber hinaus kann diese Technik bei Anwendungen mit großen Arrays und alternierendem Zugriffsmuster helfen die Anzahl von Fremdinterferenzen zu reduzieren [KW02].

6.2.2. Ändern des Speicherzugriffsverhaltens zur Optimierung der Lokalität

Will man einen Programmcode optimieren, so muss man insbesondere die Ausführungszeit oft sich wiederholender Teile, das heißt Schleifen, verringern. Durch gezielte Schleifentransformationen lässt sich sowohl die räumliche, als auch die zeitliche Lokalität steigern. In Folge dessen sinkt die Ausführungszeit eines Programms. Durch eine Schleifentransformation wird die Reihenfolge der Schleifenanweisungen verändert, die Programmsemantik bleibt dabei erhalten. Nachfolgend werden nur einige der wichtigen Techniken zum Optimieren der Laufzeit von Schleifen betrachtet. Wenn nicht anders angegeben, basiert die Beschreibung dieser Techniken auf [MSS03].

- **Schleifenvertauschung und Schleifenpermutation**. Bei mehrfach geschachtelten Schleifen kann die räumliche Lokalität durch das **Vertauschen der Reihenfolge von Schleifen** erhöht werden. Werden zwei benachbarte Schleifen vertauscht, so spricht man von einer **Schleifenvertauschung (engl. Loop Interchange)**, werden mehr als zwei Schleifen vertauscht, so spricht man von einer **Schleifenpermutation (engl. Loop Permutation)**. Die Schleifenvertauschung bzw. Permutation ist nur dann erlaubt, wenn die Datenabhängigkeiten der beteiligten Schleifen nicht verletzt werden. Dabei sind insbesondere Datenabhängigkeiten mit Richtungsfeld $(<,>)$ kritisch (mehr dazu in [SS07]).

  Die räumliche Lokalität ist am höchsten, wenn die resultierenden Speicherzugriffe sequentiell erfolgen, d. h. wenn in jedem Schleifendurchlauf auf die Datensätze zugegriffen wird, die benachbart zu den Datensätzen des vorherigen Schleifendurchlaufs sind. Zum Beschreiben des Nutzens einer Schleifentransformation in Bezug auf die räumliche Lokalität wird der Begriff **Abstand (engl. Stride)** verwendet. Die "Stride" ist der Abstand im Speicher zwischen zwei Elementen, z. B. eines Arrays, auf die in zwei aufeinander folgenden Schleifendurchläufen zugegriffen wird [Luk05].

  Als Beispiel zur Berechnung der Stride wird die folgende Schleife betrachtet:
  for $(i = 0; i < 10; i++)$ function A[i].

  In dieser Schleife wird die Variable i in jedem Schleifendurchlauf um eins erhöht. Die Stride zwischen zwei Elementen des Arrays ist 1. Wie schon in Kapitel 6.1 erwähnt, werden die Daten im Cache in Form von Cacheblöcken, bestehend aus mehreren Wörtern, gespeichert. Ist die Cachekapazität kleiner als die Größe des zu speichernden Arrays, so besteht die Gefahr, dass durch eine zu große Stride die Wörter, die bereits im Cache vorhanden sind, aus dem Cache verdrängt werden, noch bevor diese wiederverwendet werden können. Die maximale räumliche Lokalität erreicht man, wenn die Distanz zwischen zwei aufeinander folgenden Speicherzugriffen eins beträgt. Dies entspricht dem Zugriff auf die nacheinander folgenden Datensätze innerhalb der inneren Schleife. Die Technik der Schleifenvertauschung kann auch verwendet werden, um eine Vektorisierung, Parallelisierung oder auch die Wiederverwendung von Registern zu ermöglichen [MSS03].

1: **int** sum;	1: **int** sum;
2: **int** $a[i][j]$;	2: **int** $a[i][j]$;
3: **for** $(j = 0;\ j < 100;\ j++)$	3: **for** $(i = 0;\ i < 100;\ i++)$
4: **for** $(i = 0;\ i < 100;\ i++)$	4: **for** $(j = 0;\ j < 100;\ j++)$
5: $sum+ = a[i][j]$;	5: $sum+ = a[i][j]$;

Abbildung 6.1.: Loop-Interchange (Vertauschen der j-Schleife mit der i-Schleife).

Die Abbildung 6.1 zeigt das Vertauschen von zwei Schleifen anhand eines zweidimensionalen Arrays [MSS03]. Wenn man davon ausgeht, dass das Array $a[i][j]$ zeilenorientiert im Cache gespeichert ist, dann greift der Programmcode in der Abbildung 6.1 links spaltenweise und der Programmcode in der Abbildung 6.1 rechts zeilenweise auf das Array zu. Die Stride bei einem spaltenweisen Zugriff ist gleich der Anzahl der Spalten j des Arrays a. Durch das Vertauschen der i-Schleife mit der j-Schleife entsteht ein zeilenweiser Zugriff mit der Stride gleich eins. Wenn die Größe des Arrays die Größe des Caches überschreitet, so lässt sich durch eine kürzere Stride die Anzahl der Konfliktfehlzugriffe reduzieren. Bei einem Zugriff auf das Array a wird gleich ein ganzer zugehöriger Cacheblock mit mehreren Wörtern in den Cache geladen. Ist die Stride gleich eins, so werden alle Elemente des Arrays in der gleichen Reihenfolge bearbeitet, wie diese auch im Speicher abgelegt sind. Dadurch können gleich mehrere Wörter eines Cacheblocks in aufeinander folgenden Schleifeniterationen wiederverwendet werden. Bei einem spaltenweisen Zugriff auf das Array a wird bei einer großen Stride möglicherweise nur ein einzelnes Wort einer Cachzeile gelesen, bevor diese aus dem Cache verdrängt und eine neue Cachezeile geladen wird.

- **Verschmelzen von Schleifen (engl. Loop Fusion)** ist eine Transformation, bei der zwei Schleifen, die über die gleiche Schleifenvariable und gleiche Iterationsgrenzen verfügen, zu einer Schleife verschmolzen werden [MSS03]. Die Technik führt zu mehr Datenlokalität und kann die Anzahl der Konflikt- und Kapazitäts–Fehlzugriffe reduzieren. Die Schleifen können nur dann verschmolzen werden, wenn die Instruktionen der ersten Schleife von den Instruktionen der zweiten Schleife unabhängig sind. Durch das Verschmelzen von Schleifen sinkt der Schleifenoverhead, der durch das Inkrementieren von Schleifenvariablen und den Test auf das Erreichen des Schleifenlimits entsteht. Gleichzeitig kann das Verschmelzen von Schleifen die Parallelverarbeitung von Maschinenbefehlen (engl. **Instruction Level Parallelism (ILP)** ermöglichen [MSS03]. Bei ILP werden mehrere Befehle in einer mehrstufigen Pipeline parallel verarbeitet.

- **Strip-Mining und Loop-Tiling**. Unter **Strip-Minining (auch Blocking genannt)** versteht man die Partitionierung des Iterationsraumes einer einzelnen Schleife **(engl. single-nested loop)** in mehrere **Blöcke (engl. tiles)** gleicher Größe [Wol95]. Die **Blockgröße (engl. tile size)** soll dabei so gewählt werden, dass die Daten eines Blocks im Cache gespeichert werden können, um dann bei der

nachfolgenden Iteration der Schleife wiederverwendet zu werden. Ist die Blockgröße B, so wird beim Strip-Mining die Schleife i

for $(i = 0;\ i < N;\ i + +)$
  ...

durch zwei Schleifen i und j

for $(j = 0;\ j < N;\ j+ = B)$
  for $(i = j;\ i < min(N, j + B);\ i + +)$
    ....

ersetzt. Die innere Schleife i iteriert dabei über die Elemente eines Blocks der Größe $min(N - j, B)$ und die äußere Schleife j über die $\lceil N/B \rceil$ Blöcke. Wird die Strip-Mining-Technik nicht auf eine einzelne Schleife, sondern auf einen **Schleifenkomplex (engl. nested loops)** angewendet, so spricht man vom **Loop-Tiling** [Wol95].

	1: **for** $(i = 0;\ i < 100;\ i+ = 10)$
1: **for** $(j = 0;\ j < 100;\ j + +)$	2: **for** $(j = 0;\ j < 100;\ j+ = 5)$
2: **for** $(i = 0;\ i < 100;\ i{+}{+})$	3: **for** $(ii = i;\ ii < min(100, i + 10);\ ii + +)$
3: $sum+ = a[i][j];$	4: **for** $(jj = j;\ jj < min(100, i + 5);\ jj + +)$
	5: $sum+ = a[ii][jj];$

Abbildung 6.2.: Loop-Tiling der $i-$ und $j-$ Schleifen.

Die Abbildung 6.2 zeigt als Beispiel Loop-Tiling der Schleifen i und j. Die Technik des Loop-Tilings lohnt sich insbesondere für Schleifen, deren Arbeitsraum die Größe der Cache-Kapazität übersteigt.

Definition 9. *Der **Arbeitsraum (engl. working space)** eines Programmabschnittes ist die Menge aller innerhalb dieses Programmabschnittes referenzierten Datenelemente [KR07b].*

Der Erfolg von Loop-Tiling hängt stark von der gewählten Blockgröße ab. In der Literatur [LRW91, BJWE92, CM99, CM95, SSF+12] findet man eine Vielzahl von Ansätzen zur Bestimmung einer geeigneten Blockgröße (s. dazu Kapitel 8.1). Jedoch ist das Problem der Bestimmung einer geeigneten Blockgröße von der Wissenschaft nicht vollständig gelöst worden.

- **Aufrollen von Schleifen (engl. Loop Unrolling)** ist eine Optimierungstechnik, bei der der Programmcode innerhalb einer Schleife vervielfacht und dadurch die Anzahl von Schleifeniterationen reduziert wird [Muc97]. Bei einem **vollständigen Entrollen (engl. full unrolling)** wird die Schleife durch n Kopien ihres Schleifenkörpers ersetzt; n ist dabei die ursprüngliche Anzahl der Durchläufe der Schleife. Die Anzahl der Schleifenkopien wird **Aufrollfaktor (engl. unrolling**

factor) genannt [Muc97]. In dem Code in der Abbildung 6.3 wurde die Schleife i vollständig aufgerollt.

1: **for** $(i = 0;\ i < 4;\ i++)$ 2: $\quad sum+ = a[i];$	1: $sum+ = a[0];$ 2: $sum+ = a[1];$ 3: $sum+ = a[2];$ 4: $sum+ = a[3];$

Abbildung 6.3.: Vollständiges Loop Unrolling einer Schleife.

Bei einem **teilweisen Entrollen (engl. partial unrolling)** der Schleife mit dem Aufrollfaktor m, bleibt die Schleife erhalten und enthält m Kopien des Schleifenrumpfs (s. Abb. 6.4).

1: **for** $(i = 0;\ i < 4;\ i++)$ 2: $\quad sum+ = a[i];$	1: **for** $(i = 0;\ i < 4;\ i = i + 2)$ 2: $\quad sum+ = a[i];$ 3: $\quad sum+ = a[i + 1];$

Abbildung 6.4.: Teilweises Loop-Unrolling mit Aufrollfaktor 2.

Die Anzahl der Iterationen der aufgerollten Schleife ist gleich der Anzahl der Iterationen der ursprünglichen Schleife geteilt durch den Aufrollfaktor. Die Vorteile dieser Transformation können sein: der kleinere Overhead für das Initialisieren und Abfragen der Schleifen-Variablen, eine verbesserte Ausnutzung der Parallelverarbeitung auf der Ebene von Maschinenbefehlen sowie die Verbesserung der räumlichen Lokalität [FML12]. Das Aufrollen von Schleifen kann außerdem die Nutzung weiterer Optimierungstechniken, wie z. B. Scalar-Replacement, Unroll-and-Jam [HW10], Vektorisierung (s. Abschnitt 6.3), ermöglichen. Einer der Nachteile von Loop-Unrolling ist die Vergrößerung des Programmcodes, dadurch steigt die Anzahl der Instruktionen, die im Instruktionscache gespeichert werden müssen. Übersteigt die Codegröße die Größe des Instruktionscaches, so werden einzelne Programmteile aus dem Instruktionscache verdrängt und müssen nachgeladen werden, dadurch kann die Ausführungsgeschwindigkeit eines Programms sinken. Das so genannte **Register-Spilling** tritt dann auf, wenn nicht alle Variablen in Registern gespeichert werden können [FML12]. Register-Spilling entsteht, wenn durch die aufgerollte Schleife mehr Register als tatsächlich auf der Architektur vorhanden allokiert werden sollen und infolgedessen die angeforderten Daten aus dem Hauptspeicher geladen werden müssen. Damit kein Register-Spilling auftritt und die Größe des Instruktionscaches nicht überschritten wird, ist es beim Loop-Unrolling besonders wichtig zu entscheiden, welche Schleifen mit welchem Faktor aufgerollt werden sollen. Der geeignete Aufrollfaktor hängt dabei von mehreren Parametern ab: von der Größe des Instruktionscaches, der Anzahl vorhandener Register und auch der Anzahl von Schleifeniterationen. Die Compiler Intel C/C++ (ICC) und GCC unterstützen das automatische Aufrollen von Schleifen, wenn die

Optimierungsstufe `-O3` gewählt wird. Laut Intel ist es besser, das Aufrollen von Schleifen dem Compiler zu überlassen, weil sonst die manuell aufgerollten Schleifen die Auto-Vektorisierung des Programmcodes verhindern können [Gre08].

Für die ausführliche Beschreibung verschiedener Schleifen-Transformationstechniken wird auf die weiterführende Literatur [AK02, SS07, Aho08, ALSU06] verwiesen.

6.3. Vektorisierung als Möglichkeit der Optimierung bzw. Parallelisierung von rechenintensiven Schleifen

Vektorisierung (engl. Vectorization) ist eine Form der daten-parallelen Programmierung. Bei der Vektorisierung wird eine Operation gleichzeitig auf mehreren Elementen eines Vektors (eines Arrays aus Datenobjekten) ausgeführt [Gre08]. Das Prinzip des gleichzeitigen Ausführens einer Operation auf mehreren Datensätzen wird als **SIMD (Single Instruction Multiple Data)** bezeichnet. Das Konzept von SIMD steht im Gegensatz zum Konzept von **SISD (Single Instruction Single Data)**, bei dem eine Operation nur auf ein Datum angewendet werden kann.

Man kann die Vektorisierung auch als eine Art der Parallelisierung auf der Hardware-Ebene betrachten, die sich insbesondere für rechenintensive Schleifen lohnt. Die datenparallele Verarbeitung ist deshalb möglich, weil die Daten in speziellen Vektorregistern abgelegt werden. Die Daten in Vektorregistern werden dann mit Hilfe von speziellen SIMD-Instruktionen gleichzeitig verarbeitet. Die erste Erweiterung auf SIMD-Befehle waren die **MMX (Multi Media Extension)** Befehle, die von Intel bei Pentium MMX-Prozessoren 1997 verwendet wurden. Der MMX-Befehlssatz war auf 64 Bit breite Register begrenzt und unterstütze nur Integerdatentypen. Moderne Prozessoren von Intel und AMD bieten SIMD-Erweiterungen: **SSE (Streaming SIMD Extensions)** und **AVX (Advanced Vector Extensions)** [PH13, RR12a]. Der Befehlssatz **SSE (Streaming SIMD Extensions)** wurde speziell für Gleitkommazahl-Datentypen entwickelt und unterstützt die Berechnungen auf 128-Bit-Registern. Der Befehlssatz **AVX (Advanced Vector Extensions)** ist eine Erweiterung von SSE, welche es erlaubt, eine Operation auf 256-Bit-Registern durchzuführen. Die so genannten **Intrinsics**, erlauben es, die Vektorinstruktionen direkt in den Programmcode zu integrieren. Intrinsics sind Funktionsaufrufe, die prozessorspezifische Operationen in Funktionen kapseln [Wik13]. Eine Intrinsic wird vom Compiler direkt in die entsprechende Assembler-Anweisung übersetzt.

Weil die Vektorisierung mit Intrinsics viel Erfahrung erfordert, bieten die aktuellen Compiler von Intel und GCC die Möglichkeit der **automatischen Vektorisierung**, auch **Auto-Vektorisierung** genannt, an. Bei der Auto-Vektorisierung wird der Code der Schleifen vom Compiler analysiert und falls die Vektorisierung möglich ist, mit SIMD-Instruktionen, wie SSE und AVX, parallelisiert. Die Auto-Vektorisierung ist aktiv, wenn die Optimierungsstufe `-O2` oder höher gewählt wird. Der Compiler kann auch durch Angabe von Direktiven (Pragmas) bei der Auto-Vektorisierung unterstützt werden. Damit eine Schleife vektorisiert werden kann, müssen folgende Bedingungen eingehalten werden [Sab12]:

- Während der Ausführung sollte die zu vektorisierende Schleife einen festgelegten Schleifenzähler haben.
- Schleifen sollen nur eine einzige Eintritts- und eine Austrittsstelle enthalten.
- Innerhalb der zu vektorisierenden Schleife sind Switch-Anweisungen, Verzweigungen oder Sprünge nicht erlaubt. If-Anweisungen sind nur dann erlaubt, wenn diese als maskierte Anweisungen implementiert werden können, dies ist aber meistens der Fall. Bei einer maskierten If-Anweisung werden die SIMD-Operationen auf alle Elemente der Schleife angewandt, aber nur für solche Elemente gespeichert, für die die maskierte If-Anweisung erfüllt ist.
- Innerhalb einer Schleife sollen keine Funktionsaufrufe vorhanden sein.
- Nur die innerste Schleife einer verschachtelten Schleife kann vektorisiert werden.
- Datenabhängigkeiten zwischen Schleifeniterationen sollen vermieden werden, insbesondere ”read-after-write“-Abhängigkeiten.

6.4. Einschränkung der Codeoptimierung mit Hilfe eines Compilers

Die aktuellen Compiler bieten verschiedene Optimierungsstechniken, um die Laufzeit eines Programms zu verringern bzw. den benötigten Speicherplatz zu reduzieren. Die Schwierigkeit, mit der Compiler aber zu kämpfen haben, ist, dass diese nicht über das Expertenwissen eines Programmierers verfügen, dafür aber die Korrektheit des Programmcodes garantieren müssen. Compiler verwenden unterschiedliche Algorithmen zur Codeanalyse, um das Programm auf die Korrektheit nach der Anwendung von Optimierungstechniken zu untersuchen. Die Analyse des Codes führt jedoch nicht immer zu eindeutigen Ergebnissen, deshalb verzichten Compiler öfters darauf, bestimmte Codestellen zu optimieren, um die Korrektheit von Programmen zu erhalten. Inwiefern ein Compiler den Code optimieren kann, hängt auch vom Programmierstil ab. Indirekte Adressierung, Verwendung von Zeigern oder Rekursion im Programm kann die Codeanalyse für den Compiler extrem erschweren.

7. Die Anwendbarkeit des Autotuning-Verfahrens auf die sequentiellen Implementierungsvarianten expliziter PC-Verfahren vom RK-Typ

Als erster Schritt in Richtung der Entwicklung eines Autotuning-Verfahrens für die parallelen Implementierungsvarianten expliziter PC-Verfahren vom RK-Typ wird in diesem Kapitel die Möglichkeit der automatischen Selektion von sequentiellen Implementierungsvarianten während der Laufzeit untersucht. Es wird ein Grundalgorithmus des Autotunings für die sequentiellen Implementierungsvarianten vorgestellt, der eine schnelle Implementierungsvariante aus dem Kandidatenpool auswählt, dabei aber die Wahl der Programmparameter, wie z. B. geeigneter Blockgrößen, unberücksichtigt lässt. Die Anwendbarkeit des Grundalgorithmus wird anhand von Laufzeitexperimenten auf unterschiedlichen Zielarchitekturen und für verschiedene ODE-Probleme gezeigt.

Das vorliegende Kapitel ist folgendermaßen gegliedert: Zunächst wird die Berechnungsstruktur expliziter PC-Verfahren vom RK-Typ betrachtet und unterschiedliche Implementierungsvarianten zur Implementierung eines Zeitschrittes werden vorgestellt. Diese unterscheiden sich in der Schleifenorganisation und den verwendeten Datenstrukturen und folglich auch in ihrem Lokalitätsverhalten. Es folgt ein Laufzeitvergleich der verschiedenen Implementierungsvarianten. Dieser zeigt, dass die Wahl der schnellsten Implementierungsvariante von den Eigenschaften der Zielplattform, dem ODE-Problem und der Wahl des Verfahrens abhängt. Danach erfolgt die Beschreibung des Grundalgorithmus zur dynamisch Auswahl der besten Implementierungsvarianten. Die Anwendbarkeit des Grundalgorithmus wird anhand von Laufzeitexperimenten verifiziert.

7.1. Berechnungsstruktur expliziter PC-Verfahren vom RK-Typ

Es wird eine sequentielle Implementierung expliziter PC-Verfahren vom RK-Typ betrachtet. Die allgemeine Berechnungsstruktur expliziter PC-Verfahren vom RK-Typ zeigt der Pseudocode in Abbildung 7.1.

Das Rahmenprogramm enthält eine Schleife, die über das Integrationsintervall $[t_0, t_e]$ läuft. In jedem Schleifendurchlauf wird die Funktion *compute_time_step*$(t, h, \boldsymbol{\eta}_{old})$ aufgerufen und anschließend die Schrittweitenkontrolle durchgeführt. Die Funktion *compute_time_step*$(t, h, \boldsymbol{\eta}_{old})$ führt einen Zeitschritt entsprechend der Berechnungsvorschrift (3.27) aus. Nach jedem Zeitschritt findet eine Schrittweitenkontrolle durch die Funktion *stepsize_control*$(h, \boldsymbol{\eta}, \hat{\boldsymbol{\eta}})$ statt. In der Schrittweitenkontrolle wird der lokale Fehler ϵ sowie die neue Schrittweite h_{new} berechnet. Ist der lokale Fehler ϵ klein genug, so wird der Zeitschritt und die berechnete Approximation akzeptiert und der nächste Zeitschritt mit

```
    // Initialisiere t und h
 1: t ← t_0; // Setze t auf Anfang des Integrationsintervalls t_0
 2: h ← h_init;
 3: η ← y_0;                                   // Initialisiere η mit Startwert y_0
 4: while (t < t_e) // Durchlaufe das Integrationsintervall [t_0, t_e]
 5: {
 6:     η_old ← η;
 7:     (η, η̂) ← compute_time_step(t, h, η_old);
 8:     (h_new, ε) ← stepsize_control(h, η, η̂);
 9:     if (wenn ε klein genug) t ← t + h;       // Zeitschritt wird akzeptiert
10:     else η ← η_old;                          // Zeitschritt wird verworfen
11:     aktualisiere t, wenn ε klein genug;
12:     h ← min(h_new, t_e − t);      // h darf das Intervallende nicht überschreiten
13: }
```

Abbildung 7.1.: Pseudocode einer Zeitschrittfunktion expliziter PC-Verfahren vom RK-Typ.

der neuen Schrittweite ausgeführt. Ansonsten wird der gleiche Zeitschritt nochmal mit der geänderten Schrittweite wiederholt.

7.2. Kandidatenpool sequentieller Implementierungsvarianten

Die Berechnungsstruktur expliziter PC-Verfahren vom RK-Type erlaubt die Erzeugung einer Vielzahl von Implementierungsvarianten, die zwar das gleiche Rahmenprogramm besitzen, aber in der Implementierung der Zeitschrittfunktion *compute_time_step*$(t, h, \boldsymbol{\eta}_{old})$ voneinander abweichen. Die verschiedenen Varianten der Zeitschrittfunktion können mit Hilfe der bereits beschriebenen Optimierungstechniken, wie Schleifenvertauschung und Schleifenverschmelzung, sowie Loop-Tiling, realisiert werden. Die so erzeugten Varianten unterscheiden sich zwar in der Reihenfolge verwendeter Schleifen und den Datenstrukturen, sind jedoch numerisch äquivalent. Aufgrund unterschiedlicher Schleifenstruktur besitzen die Implementierungsvarianten ein unterschiedliches Lokalitäts- und Speicherzugriffsverhalten. Welche der Implementierungsvarianten mit welchen Parametern, z. B. Blockgrößen, die schnellste ist, hängt von den Eigenschaften der Zielplattform, wie der Architektur und der Assoziativität des Caches, und dem zu lösenden ODE-Problem ab.

Existierende Autotuning-Systeme benutzen häufig Source-to-Source-Compiler, wie z. B. ROSE [QL11] oder CHILL [Che07], um Schleifentransformationen automatisch durchführen zu können. Infolge der vorangegangenen Arbeit von Matthias Korch [KR07b] waren die sequentiellen Implementierungsvarianten des betrachteten PC-Verfahrens aber bereits verfügbar. Für die Evaluation der Anwendbarkeit musste daher kein Source-to-Source-Compiler zur Erzeugung von Implementierungsvarianten verwendet werden. Die in [KR07b] entwickelten allgemeinen und spezialisierten Implementierungsvarian-

ten konnten direkt zur Bildung einer Auswahlmenge von Implementierungsvarianten für den Autotuner verwendet werden. Die Menge aller Implementierungsvarianten wird im Folgenden als Kandidatenpool bezeichnet. Als nächstes werden die Implementierungsvarianten des Kandidatenpools zum Verständnis der Funktionsweise des implementierten selbstadaptiven Solvers näher beschrieben.

7.2.1. Unterschiedliche Möglichkeiten der Realisierung einer Zeitschrittfunktion

Es gibt verschiedene Möglichkeiten zur Realisierung einer Zeitschrittfunktion. Die Implementierung eines Zeitschrittes basiert auf der Berechnungsvorschrift (3.27). Diese führt uns zu einer perfekt geschachtelten Schleife der Dimension 4. In der Schleife wird über

- die Anzahl der Korrektorschritte k mit $k = 1, \dots, m$,
- die Argumentvektoren $\mathbf{Y}_l^{(k)}$ mit $l = 1, \dots, s$,
- die Summanden $\sum_{i=1}^{s} a_{li}\mathbf{F}_i^{(k-1)}$ mit $i = 1, \dots, s$
- und die Systemdimension $j = 1, \dots, n$

iteriert.

Im allgemeinen, werden zur Berechnung einer Komponente $\mathbf{f}_j\left(t_\kappa + c_i h_\kappa, \mathbf{Y}_i^{(k-1)}\right)$ alle Komponenten des Argumentvektors $\mathbf{Y}_i^{(k-1)}$ gebraucht. Dies bedeutet, dass die Berechnung jeder Komponente des Argumentvektors $Y_{l,j}^{(k)}$ von den Argumentvektoren des vorherigen Korrektorschrittes $\mathbf{Y}_1^{(k-1)}, \dots, \mathbf{Y}_s^{(k-1)}$ abhängt. Da die Korrektorschritte sequentiell voneinander abhängen, müssen diese nacheinander berechnet werden. Die Schleife k, welche über die Anzahl der Korrektorschritte läuft, kann deshalb nur als die äußerste Schleife, die die Schleifen l, i und j umfasst, implementiert werden. Die inneren Schleifen l, i und j sind voneinander unabhängig und bilden einen voll permutierbaren Schleifenkomplex. Auf einen Schleifenkomplex, der voll permutierbar ist, können unterschiedliche Codeoptimierungsmethoden angewandt werden. Die inneren Schleifen des Schleifenkomplexes können vertauscht, verschmolzen, aufgesplittet oder aufgerollt werden. Außerdem kann Loop-Tiling auf alle l, i und j Schleifen angewandt werden. Für eine spezielle Klasse von Anfangswertproblemen jedoch, nämlich für solche, deren Komponentenfunktion $\mathbf{f}_j$ nur wenige Komponenten des Argumentvektors $\mathbf{Y}_i^{(k-1)}$ benutzt, um $\mathbf{f}_j\left(t_\kappa + c_i h_\kappa, \mathbf{Y}_i^{(k-1)}\right)$ auszurechnen (s. Kapitel 7.2.3), kann zusätzlich die k Schleife mit der Schleife j vertauscht werden. Die Optimierungstechniken, die für die Erzeugung von Implementierungsvarianten benutzt werden, sind bereits im Kapitel 6.2.2 eingeführt worden.

7.2.2. Allgemeine Implementierungsvarianten

Je nach Struktur des zugrunde liegenden Differentialgleichungssystems wird zur Berechnung einer Komponente $\mathbf{f}_j(t, \mathbf{y})$ entweder auf alle oder nur auf eine Teilmenge der Kom-

ponenten des Argumentvektors $\mathbf{y}$ zugegriffen. Die **allgemeinen** Implementierungsvarianten sind anwendbar auf ODE-Probleme, bei denen die Funktion $\mathbf{f}(t, \mathbf{y})$ ein beliebiges Zugriffsmuster aufweisen darf. Da das Zugriffsmuster der Funktion $\mathbf{f}(t, \mathbf{y})$ so sein kann, dass zur Berechnung einer Komponente $\mathbf{f}_j(t, \mathbf{y})$ der Zugriff auf alle Komponenten von $\mathbf{y}$ erfolgt, muss die Berechnung der Korrektorschritte sequentiell erfolgen. Als Konsequenz muss die k-Schleife bei allen allgemeinen Implementierungsvarianten die äußerste Schleife bleiben.

Betrachten wir die Berechnungen eines Korrektorschrittes, so sehen wir, dass in jedem Korrektorschritt k nur die Daten des aktuellen Korrektorschrittes k sowie die Daten des vorherigen Korrektorschrittes $k-1$ gebraucht werden. Deshalb ist es ausreichend, zur Berechnung eines Korrektorschrittes maximal vier $s \times n$ Matrizen:

- $F^{(\text{prev})} = \left(\mathbf{F}_1^{(k-1)}, \dots, \mathbf{F}_s^{(k-1)}\right)$,
- $Y^{(\text{prev})} = \left(\mathbf{Y}_1^{(k-1)}, \dots, \mathbf{Y}_s^{(k-1)}\right)$,
- $F^{(\text{cur})} = \left(\mathbf{F}_1^{(k)}, \dots, \mathbf{F}_s^{(k)}\right)$,
- $Y^{(\text{cur})} = \left(\mathbf{Y}_1^{(k)}, \dots, \mathbf{Y}_s^{(k)}\right)$.

im Speicher zu halten und nach jeder Iteration der k-Schleife die Zeiger der (prev) und (cur) Matrizen zu vertauschen.

Betrachtet man die Berechnungsvorschrift (3.27), so stellt man fest, dass zur Implementierung des Verfahrens auch weniger als vier der oben erwähnten Matrizen ausreichen. Entweder genügt es, die zwei Matrizen $F^{(\text{prev})}$ und $F^{(\text{cur})}$ im Speicher zu halten, dann braucht man noch einen zusätzlichen Vektor $\mathbf{Y}$, um alle Funktionsargumente $F^{(\text{cur})}$ anhand von $F^{(\text{prev})}$ in dem aktuellen Korrektorschritt auszurechnen. Diese Möglichkeit wurde bei den Implementierungsvarianten A und E umgesetzt. Eine weitere Möglichkeit, die bei allen von der Implementierung D abgeleiteten Varianten realisiert wurde, ist es, die Matrizen $Y^{(\text{prev})}$ und $Y^{(\text{cur})}$ im Speicher zu halten und zusätzlich noch eine skalare Variable F zu verwenden, um alle Funktionsauswertungen für die Berechnung von $Y^{(\text{cur})}$ zwischenzuspeichern.

Zusätzlich zu den bereits erwähnten Vektoren werden noch zwei Approximationsvektoren $\Delta\boldsymbol{\eta}^{(m)}$ und $\Delta\boldsymbol{\eta}^{(m-1)}$ der Dimension n benötigt. Den Vektor $\Delta\boldsymbol{\eta}^{(m)} := \boldsymbol{\eta}_{\kappa+1} - \boldsymbol{\eta}_\kappa$ braucht man, um $\boldsymbol{\eta}_{\kappa+1} = \boldsymbol{\eta}_\kappa + \Delta\boldsymbol{\eta}^{(m)}$ auszurechnen. Mit Hilfe des zweiten Vektors $\Delta\boldsymbol{\eta}^{(m-1)} := \hat{\boldsymbol{\eta}}_{\kappa+1} - \boldsymbol{\eta}_\kappa$ schätzt man den lokalen Fehler $\epsilon = \|\boldsymbol{\eta}_{\kappa+1} - \hat{\boldsymbol{\eta}}_{\kappa+1}\| = \|\Delta\boldsymbol{\eta}^{(m)} - \Delta\boldsymbol{\eta}^{(m-1)}\|$ ab.

Die Tabelle 7.1 gibt eine Übersicht der im Kandidatenpool enthaltenen allgemeinen Implementierungsvarianten. Diese Varianten wurden in [KR07b] präsentiert. Die in der Tabelle 7.1 enthaltenen Implementierungsvarianten unterscheiden sich in der Schleifenstruktur der Zeitschrittfunktion compute_time_step(t) und den verwendeten Datenstrukturen. Als nächstes werden die einzelnen allgemeinen Implementierungsvarianten näher beschrieben.

Tabelle 7.1.: Allgemeine Implementierungsvarianten.

Implementierung	Datenstrukturen	Schleifenstruktur	Eigenschaften
A	$\mathbf{F}_1^{(k)}, \ldots, \mathbf{F}_s^{(k)} \in \mathbb{R}^n$, $\mathbf{F}_1^{(k-1)}, \ldots, \mathbf{F}_s^{(k-1)} \in \mathbb{R}^n$, $\mathbf{Y} \in \mathbb{R}^n$	k–l–i–j	vektororientiert: innere Schleifen iterieren über die Systemdimension; große räumliche Lokalität
E	$\mathbf{F}_1^{(k)}, \ldots, \mathbf{F}_s^{(k)} \in \mathbb{R}^n$, $\mathbf{F}_1^{(k-1)}, \ldots, \mathbf{F}_s^{(k-1)} \in \mathbb{R}^n$, $\mathbf{Y} \in \mathbb{R}^n$	k–l–j–i	Ausnutzung der zeitlichen Lokalität der i-Schleife bez. Schreibzugriffe auf die Argumentvektoren
D	$\mathbf{Y}_1^{(k)}, \ldots, \mathbf{Y}_s^{(k)} \in \mathbb{R}^n$, $\mathbf{Y}_1^{(k-1)}, \ldots, \mathbf{Y}_s^{(k-1)} \in \mathbb{R}^n$ $\mathbf{F} \in \mathbb{R}^1$	k–i–j–l	Ausnutzung der zeitlichen Lokalität der l-Schleife bez. Lesezugriffe auf die Ergebnisse von Funktionsauswertungen
Dblock	$\mathbf{Y}_1^{(k)}, \ldots, \mathbf{Y}_s^{(k)} \in \mathbb{R}^n$, $\mathbf{Y}_1^{(k-1)}, \ldots, \mathbf{Y}_s^{(k-1)} \in \mathbb{R}^n$, $\mathbf{F} \in \mathbb{R}^B$	k–i–j–l–jj	ähnlich zu Implementierung D, aber zusätzlich Loop-Tiling der j-Schleife und Vertauschen mit der l-Schleife
PipeDe2m	$\mathbf{Y}_1^{(k)}, \ldots, \mathbf{Y}_s^{(k)} \in \mathbb{R}^n$, $\mathbf{Y}_1^{(k-1)}, \ldots, \mathbf{Y}_s^{(k-1)} \in \mathbb{R}^n$	k–j–i–l	basiert auf D; j Schleife iteriert über die Schleifen l und i; nutzt zeitliche Lokalität der Schleifen i und l
PipeDb2m	$\mathbf{Y}_1^{(k)}, \ldots, \mathbf{Y}_s^{(k)} \in \mathbb{R}^n$, $\mathbf{Y}_1^{(k-1)}, \ldots, \mathbf{Y}_s^{(k-1)} \in \mathbb{R}^n$	k–j–i–jj–l	ähnlich zu *PipeDe2m*, aber zusätzlich Loop-Tiling der j-Schleife und Vertauschen mit der i-Schleife
PipeDb2mt	$\mathbf{Y}_1^{(k)}, \ldots, \mathbf{Y}_s^{(k)} \in \mathbb{R}^n$, $\mathbf{Y}_1^{(k-1)}, \ldots, \mathbf{Y}_s^{(k-1)} \in \mathbb{R}^n$, $\mathbf{F} \in \mathbb{R}^B$	k–j–i–(jj)–l–jj	ähnlich zu *PipeDb2m*, aber das Loop-Tiling ist auf die l-Schleife ausgeweitet

```
    // Funktionsauswertung des trivialen Prädiktors η = η_κ

 1: for (l = 0; l < s; l++)
 2:     for (j = 0; j < n; j++) F_lj^(cur) = f_j(t + c_l h, η);

    // m Korrektorschritte

 3: for (k = 0; k < m; k++)
 4: {
 5:     SWAP(F^(cur), F^(prev));
 6:     for (l = 0; l < s; l++)
 7:     {
 8:         for (j = 0; j < n; j++) Y_j = a_l0 F_0j^(prev);
 9:         for (i = 1; i < s; i++)
10:             for (j = 0; j < n; j++) Y_j += a_li F_ij^(prev);
11:         for (j = 0; j < n; j++) Y_j = η_j + h Y_j;
12:         for (j = 0; j < n; j++) F_lj^(cur) = f_j(t + c_j h, Y);
13:     }
14: }
    // Berechnung des Fehlervektors e = η_{κ+1} − η̂_{κ+1} und Δη^(m) = Σ_{i=1}^{s} b_i F_i^(m)

15: for (j = 0; j < n; j++)
16: {
17:     Δη_j^(m) = b_0 F_0j^(cur);
18:     e_j = b_0 (F_0j^(cur) − F_0j^(prev));
19: }

20: for (i = 1; i < s; i++)
21:     for (j = 0; j < n; j++)
22:     {
23:         Δη_j^(m) += b_i F_ij^(cur);
24:         e_j += b_i (F_ij^(cur) − F_ij^(prev));
25:     }

    // Berechnung der Approximation η_{κ+1} durch die Aktualisierung von η

26: for (j = 0; j < n; j++) η_j += h Δη_j^(m);
```

Abbildung 7.2.: Pseudocode eines Zeitschrittes der Implementierungsvariante A.

- Die Abbildung 7.2 zeigt die Implementierung der Zeitschrittfunktion der Implementierungsvariante A. Die Implementierung A ist eine vektororientierte Implementierung. Die n Vektoren $\mathbf{F}_i^{(k-1)}$ und $\mathbf{Y}_l^{(k)}$ werden innerhalb der inneren Schleifen berechnet. Alle inneren Schleifen iterieren über die Systemdimension $j = 1, \ldots, n$, die äußerste Schleife iteriert über die Argumentvektoren $\mathbf{Y}_l^{(k)}$ mit $l = 1, \ldots, s$. Die Implementierung A nutzt die räumliche Lokalität aus, da auf alle Vektoren elementweise mit aufsteigendem Index zugegriffen wird. Der Nachteil der Implementierung A ist die geringe zeitliche Lokalität. Überschreitet die Gesamtheit aller in einem Zeitschritt verwendeten Vektoren die Cachegröße, so werden die Teilbereiche der Vektoren im Cache überschrieben. Findet in der nächsten Schleifeniteration ein erneuter Speicherzugriff auf die Elemente überschriebener Vektoren statt, so müssen diese erneut in den Cache geladen werden, was erneut zur Verdrängung der im Cache gespeicherter Vektorelemente führen kann.

- Die Implementierung E wurde mit dem Ziel entwickelt, die zeitliche Lokalität bezüglich der Schreibzugriffe auf die Argumentvektoren $\mathbf{Y}_l^{(k)}$ auszunutzen [KR07b]. Die Argumentvektoren $\mathbf{Y}_l^{(k)}$ entsprechen der Summe $\sum_{i=1}^{s} a_{li}\mathbf{F}_i^{(k-1)}$. Die Implementierung E verwendet die gleichen Vektoren wie die Implementierung A. Die i-Schleife in Zeile (9) von Abbildung 7.2 wurde jedoch mit der j-Schleife in Zeile (10) vertauscht. Die innerste Schleife läuft jetzt über die Stufen i. Weiter wurden alle Schleifen, die über die Systemdimension $j = 1, \ldots, n$ iterieren, miteinander verschmolzen.

- Die Implementierung D ist entstanden als Folge von mehreren Schleifenvertauschungen und mehrmaligem Verschmelzen von Schleifen mit dem Ziel, eine zeitliche Wiederverwendung der Funktionsauswertungen für die Berechnung von $Y^{(\mathrm{cur})}$ zu ermöglichen. Für diese Implementierung war eine Veränderung der Datenstrukturen notwendig.

  In der Implementierung D werden zwei Matrizen $Y^{(\mathrm{prev})}$ und $Y^{(\mathrm{cur})}$ und zusätzlich noch eine skalare Variable F verwendet, um alle Funktionsauswertungen für die Berechnung von $Y^{(\mathrm{cur})}$ zwischenzuspeichern. Die innerste Schleife dieser Implementierungsvariante bildet die l-Schleife. Dies erlaubt eine zeitnahe Wiederverwendung der Funktionsauswertung einer Komponente $f_j\left(t_\kappa + c_l h_\kappa, \mathbf{Y}_i^{(k-1)}\right)$ für das Berechnen der entsprechenden Komponenten der Argumentvektoren $Y_{l,j}^{(k)}$, $l = 1, \ldots, s$. Der Pseudocode der Implementierung D ist in der Abbildung 7.3 gegeben.

  Weitere allgemeine Implementierungen, wie *Dblock*, *PipeDe2m*, *PipeDb2m* und *PipeDb2mt*, leiten sich alle von der allgemeinen Struktur der Implementierung D ab.

- Die Implementierung *Dblock* besitzt die gleiche Schleifenstruktur wie die Implementierung D. Um eine bessere Ausnutzung der räumlichen und zeitlichen Lokalität zu ermöglichen, wurde bei der Implementierung *Dblock* zusätzlich die Optimierungstechnik des Loop-Tilings (s. Kapitel 6.2.2) eingebaut. In der Implementierung D wurde das Loop-Tiling der Schleife j innerhalb der Schleife l implementiert. Die

```
    // m Korrektorschritte und Berechnung von Δη^(m−1) = Σ_{i=1}^{s} b_i F_i^(m−1)

 1: for (k = 0; k < m; k++)
 2: {
 3:     SWAP(Y^(cur), Y^(prev));
 4:     for (j = 0; j < n; j++)
 5:     {
 6:         F = h f_j (t + c_0 h, (k == 0 ? η : Y_0^(prev)));
 7:         for (i = 1; i < s; i++) Y_{i,j}^(cur) = η_j + a_{i0} F;
 8:         if (k == m − 1) Δη_j^(m−1) = b_0 F;
 9:     }
10:     for (l = 1; l < s; l++)
11:         for (j = 0; j < n; j++)
12:         {
13:             F = h f_j (t + c_l h, (k == 0 ? η : Y_l^(prev)));
14:             for (i = 1; i < s; i++) Y_{i,j}^(cur) += a_{il} F;
15:             if (k == m − 1) Δη_j^(m−1) += b_l F;
16:         }
17: }
    // Berechnung von Δη^(m) = Σ_{i=1}^{s} b_i F_i^(m)

18: SWAP(Y^(cur), Y^(prev));
19: for (j = 0; j < n; j++)
20:     Δη_j^(m) = b_0 h f_j (t + c_0 h, Y_0^(prev));
21: for (l = 1; l < s; l++)
22:     for (j = 0; j < n; j++)
23:         Δη_j^(m) += b_l h f_j (t + c_l h, Y_l^(prev));


    // Berechnung von e = η_{κ+1} − η̂_{κ+1} und η_{κ+1} durch die Aktualisierung von
    // η

24: for (j = 0; j < n; j++)
25: {
26:     e_j = Δη_j^(m) − Δη_j^(m−1);
27:     η_j += Δη_j^(m);
28: }
```

Abbildung 7.3.: Pseudocode eines Zeitschrittes der Implementierungsvariante *D*.

Zeilen (4)-(9) in der Implementierung D, Abbildung 7.3, bilden die erste Iteration der l-Schleife. Der Iterationsraum, der durch die Schleifen l und j gebildet wird, ist in Blöcke der Blockgröße B aufgeteilt worden. Zunächst wurden die Schleifen j in den Zeilen (4), (11), (19) und (22), der Form

for $(j = 0;\ j < n;\ j{+}{+})$

durch zwei ineinander geschachtelte Schleifen j und jj

for $(j = 0;\ j < n - B + 1;\ j{+} = B)$
  for $(jj = 0;\ jj < B;\ jj{+}{+})$

ersetzt. Danach wurde die Schleife jj von der Schleife j getrennt und repliziert, damit alle Berechnungen innerhalb der Schleife jj ausgeführt werden. Um alle Funktionsauswertungen für die Berechnung von $Y^{(\mathrm{cur})}$ zwischenzuspeichern, wird anstelle einer skalaren Variable F nun ein Vektor $\mathbf{F}$ der Blockgröße B verwendet.

Als Beispiel für das Anwenden des Loop-Tiling wird die verschachtelte Schleife l in den Zeilen (10)-(17), Abbildung 7.3, betrachtet. Nach Anwenden des Loop-Tilings sieht die l-Schleife wie in der Abbildung 7.4 aus. Man sieht, dass bei der Programmvariante *Dblock* alle Berechnungen innerhalb der jj-Schleife ausgeführt werden.

for $(l = 1;\ l < s;\ l{+}{+})$
  for $(j = 0;\ j \leq n - B + 1;\ j{+} = B)$
  {
    for $(jj = 0;\ jj < B;\ jj{+}{+})$
    $\mathbf{F}_{jj} = hf_{j+jj}\Big(t + c_l h, \Big(k == 0\ ?\ \boldsymbol{\eta} : \mathbf{Y}_l^{(\mathrm{prev})}\Big)\Big);$
    for $(i = 1;\ i < s;\ i{+}{+})$
      for $(jj = 0;\ jj < B;\ jj{+}{+})\ Y_{i,j+jj}^{(\mathrm{cur})} \mathrel{+}= a_{il}\mathbf{F}_{jj};$
    if $(k == m - 1)$
    for $(jj = 0;\ jj < B;\ jj{+}{+})\ \Delta\eta_j^{(m-1)} \mathrel{+}= b_l \mathbf{F}_{jj};$
  }

Abbildung 7.4.: Loop-Tiling der l-Schleife bei der Implementierungsvariante *Dblock*.

- Die Implementierung *PipeDe2m* basiert auf der Implementierung D, die Schleifen j und i wurden aber miteinander vertauscht. Die Schleife j enthält jetzt die Schleifen i und l, beide iterieren über die Anzahl der Stufen s. Diese Programmvariante nutzt die zeitliche Lokalität aus durch eine Wiederverwendung der Funktionsauswertungen zum zeitnahen Aktualisieren der Komponenten von $Y^{(\mathrm{cur})}$ innerhalb der i und l Schleifen.

- Die Implementierung *PipeDb2m* besitzt eine ähnliche Schleifenstruktur wie die Implementierung *PipeDe2m*. Bei dieser Implementierung wurde zusätzlich Loop-

Tiling der j-Schleife mit Vertauschen der i-Schleife implementiert, was zu einer Steigerung der räumlichen Lokalität gegenüber der Implementierung *PipeDe2m* führt.

- Die Implementierung *PipeDb2mt* geht von der Schleifenstruktur der Implementierung *PipeDb2m* aus. Zur Verbesserung der räumlichen Lokalität wurde Loop-Tiling der j-Schleife von der i-Schleife auf die l-Schleife ausgeweitet.

Am effizientesten würden alle Implementierungsvarianten arbeiten, wenn alle verwendeten Datenstrukturen im Cache gespeichert werden können. Die Gesamtheit aller von einer Implementierung verwendeten Datenstrukturen entspricht dem Arbeitsraum einer Implementierungsvariante (vgl. Definition 9). Da der Cache eine begrenzte Größe hat, passt der Arbeitsraum einer Implementierungsvariante nicht immer in den Cache hinein. Übersteigt der Arbeitsraum die Cachegröße, so fallen Teile des Arbeitsraumes aus dem Cache heraus und müssen aus dem Hauptspeicher geholt werden. Dadurch steigt die Anzahl der Kapazitätsfehlzugriffe an und die Effizienz der Implementierung nimmt in Folge des langsamen Zugriffs auf den Hauptspeicher ab.

Zur Bestimmung eines Arbeitsraumes eines Schleifenkomplexes bestimmt man zuerst die Menge der Datenelemente, die innerhalb der innersten Schleifen des Komplexes referenziert werden. Danach bildet man die Vereinigung dieser Menge mit der Menge der Datenelemente, die in umschließenden Schleifen referenziert werden.

Den wichtigsten und größten Arbeitsraum einer Implementierungsvariante stellt der Arbeitsraum eines Zeitschrittes WS_{ts} dar. Wird die Schrittweitenkontrolle als ein Teil des Zeitschrittes aufgefasst, so enthält der Arbeitsraum eines Zeitschrittes alle im Programm referenzierten Datenelemente und entspricht somit dem Gesamtspeicherplatz einer Implementierungsvariante. Betrachtet man die Tabelle 7.1, so wird deutlich, dass die allgemeinen Implementierungsvarianten in jedem Korrektorschritt zwei Matrizen der Größe $s \times n$ speichern müssen. Die allgemeinen Implementierungsvarianten mit Loop-Tiling müssen zusätzlich noch einen Vektor der Größe B speichern, die Implementierungsvarianten A und E benötigen einen zusätzlichen Vektor $\mathbf{Y}$ der Größe n. Zwei weitere Vektoren $\Delta\boldsymbol{\eta}^{(m)}$ und $\Delta\boldsymbol{\eta}^{(m-1)}$ werden von allen Implementierungsvarianten zur Durchführung der Schrittweitenkontrolle gebraucht. Zu Beginn jedes Zeitschrittes greifen alle Implementierungen auf den Approximationsvektor $\boldsymbol{\eta}$ der Größe n zu, dieser repräsentiert den trivialen Prädiktor $\boldsymbol{\eta} = \boldsymbol{\eta}_\kappa$. Wird der Speicherplatzbedarf für die Verfahrenskoeffizienten und die verwendeten skalaren Variablen vernachlässigt, weil diese nur wenig Speicherplatz benötigen [Kor07], so ist die Größe des Arbeitsraumes eines Zeitschrittes bei allgemeinen Implementierungsvarianten von der Stufenanzahl s und der Systemgröße n abhängig. Bei Implementierungsvarianten mit Loop-Tiling hängt die Größe des Arbeitsraumes eines Zeitschrittes zusätzlich von der gewählten Blockgröße B ab. Die Größe des Arbeitsraumes eines Zeitschrittes kann bei allgemeinen Implementierungsvarianten *D*, *PipeDe2m*, *PipeDb2m* durch die Formel $WS_{ts} = (2s+3) \cdot n$, bei den Loop-Tiling Varianten *Dblock*, *PipeDb2mt* durch die Formel $WS_{ts} = (2s+3) \cdot n + B$ ausgedrückt werden. Die Größe des Arbeitsraumes bei den Implementierungsvarianten A und E ist durch die Formel $WS_{ts} = (2s+4) \cdot n$ gegeben.

Außer dem Arbeitsraum eines Zeitschrittes können innerhalb einer Implementierungsvariante noch weitere Arbeitsräume bestimmt werden, wie z. B. der Arbeitsraum eines Korrektorschrittes. Mehr dazu im Kapitel 8.4.2. Ob und wie diese Arbeitsräume die Laufzeit der Implementierung beeinflussen, hängt u. U. von den Cacheparametern der Zielplattform, dem ODE-Problem und dem verwendeten Compiler ab.

7.2.3. Die spezialisierten Implementierungsvarianten

Alle allgemeinen Implementierungsvarianten haben gemeinsam, dass die äußerste Schleife k innerhalb eines Zeitschrittes über die Anzahl der Korrektorschritte läuft. Dies bedeutet, dass bei allen allgemeinen Implementierungsvarianten die Berechnung der Korrektorschritte sequentiell erfolgen muss. Die inneren Schleifen des Schleifenkomplexes iterieren entweder über die Systemdimension j oder über die Anzahl der Stufen mit den Schleifen i und l. Für eine spezielle Klasse von Anfangsproblemen jedoch, deren Jacobi-Matrix für die Funktion $\mathbf{f}$ eine **Bandstruktur** aufweist [MCP08, WGS09], kann eine Anordnung der ODE-Komponenten gefunden werden, so dass zum Berechnen einer Funktionskomponente f_j nur ein Teil der Elemente des Argumentvektors $\mathbf{y}$ gebraucht wird.

Matrizen mit Bandstruktur enthalten nur in einem relativ schmalen Streifen in der Umgebung der Hauptdiagonalen von Null verschiedene Elemente und entstehen z. B. bei der Diskretisierung von partiellen Differentialgleichungssystemen mit Hilfe von finiten Differenzen.

Definition 10. *Seien $p, q \in \mathbb{N}$ mit $p, q \geq 0$. Die Matrix $A \in \mathbb{R}^{n,n}$ mit $A = (a_{ij})$ hat eine Bandstruktur mit der Bandbreite $l = p + q + 1$, falls $a_{ij} = 0$ für $j + p < i$ und $i + q < j$ gilt [SK11].*

Matrizen mit Bandstruktur sind Spezialfälle von **dünnbesetzten (engl. sparse)** Matrizen. Die Bandbreite einer Matrix kann durch eine Umnummerierung der Gitterpunkte minimiert werden [Sue03]. Das Problem der Umnummerierung der Gitterpunkte zur Reduktion der Bandbreite kann auf das Problem der **Bandbreitenminimierung einer Matrix (engl. Matrix Bandwidth Minimization)** zurückgeführt werden. Bei dem Problem der Bandbreitenminimierung wird nach einer Permutation der Indizes einer Matrix gesucht, so dass eine Bandmatrix minimaler Bandbreite entsteht. Dieses Problem ist NP-schwer [Pap76]. Zur Bandbreitenminimierung werden daher spezielle Algorithmen verwendet, die nach einer Permutation der Indizes einer Matrix suchen, die eine Bandbreite in der Nähe der minimalen Bandbreite liefern [CM69]. Die Algorithmen zur Bandbreitenminimierung stellen die von Null verschiedenen Elemente einer Matrix als Knoten eines Graphen dar und suchen nach einer Umnummerierung der Knoten, die zu einer geringeren Bandbreite dieser Matrix führt. Der erste Algorithmus zur Bandbreitenminimierung einer Matrix, im Jahr 1969 vorgeschlagen, geht auf Cuthill-McKee [CM69] zurück. In [PnPCM04] findet man eine Übersicht unterschiedlicher Algorithmen zur Bandbreitenminimierung. Einige der aktuellen Publikationen zur Bandbreitenminimierung sind in [MCP08, WGS09] zu finden.

Wenn die Jacobi-Matrix eine Bandstruktur aufweist, so befinden sich die Komponenten, die für die Funktionsauswertung der Komponente f_j gebraucht werden, in einem kleinen Bereich um den Index j (s. Abb. 7.5). Die **Zugriffsdistanz** ist ein Maß, das beschreibt, wie nah die Komponenten des Argumentvektors $\mathbf{y}$ liegen, die für die Funktionsauswertungen $\mathbf{f}_j(t, \mathbf{y})$ gebraucht werden [Kor07].

Definition 11. *Die Zugriffsdistanz einer Funktion* $\mathbf{f}$*, als* $d(\mathbf{f})$ *bezeichnet, ist der kleinste Wert* b*, so dass alle Funktionsauswertungen* $\mathbf{f}_j(t, \mathbf{y})$ *nur auf einen Teil*

$$\{y_{j-b}, \ldots, y_j, \ldots, y_{j+b}\}$$

der Komponenten von y *zugreifen [Kor07].*

Aus der Definition 11 folgt, dass die Berechnung der Funktionskomponente $\mathbf{f}_j(t, \mathbf{y})$ von Komponenten y_i mit $|i - j| > b$ unabhängig ist [Kor07]. Dies bedeutet, dass alle Einträge der Jacobi-Matrix $a_{ij} = \frac{\partial f_j(t,\mathbf{y})}{\partial y_i}$ mit $|i - j| > b$ gleich Null sind. Somit besitzt die Jacobi-Matrix $\mathbf{f}_\mathbf{y}(t, \mathbf{y})$ für die Funktion $\mathbf{f}$ nach Definition 10 eine Bandstruktur mit der Bandbreite $D = 2d(\mathbf{f}) + 1$ [Kor07]. Als nächstes folgt die Definition der beschränkten Zugriffsdistanz.

Definition 12. *Die Funktion der rechten Seite* $\mathbf{f}$ *einer ODE mit der Systemgröße* n *besitzt eine* ***beschränkte Zugriffsdistanz****, wenn gilt* $d(\mathbf{f}) \ll n$ *[Kor07].*

Die Anforderung $d(\mathbf{f}) \ll n$ an die Zugriffsdistanz bedeutet, dass $d(\mathbf{f})$ um eine Größenordnung kleiner als die Systemgröße n sein sollte. Zum Beispiel besitzt das Testproblem BRUSS2D-MIX (mehr Details im Anhang A.1) eine beschränkte Zugriffsdistanz. Das Testproblem BRUSS2D-MIX entstand durch die Diskretisierung einer zeitabhängigen partiellen Differentialgleichung mit Hilfe der Linienmethode über einen NxN Gitter mit der Gittergröße $1/(N - 1)$. Das resultierende ODE-System für BRUSS2D-MIX hat die Systemgröße $2N^2$ und die beschränkte Zugriffsdistanz $2N$.

Bei den allgemeinen Implementierungsvarianten werden für die Funktionsauswertung einer Komponente f_j alle Komponenten des Argumentvektors $\mathbf{y}$ gebraucht. Besitzt eine ODE dagegen eine beschränkte Zugriffsdistanz, so werden für die Funktionsauswertung einer Komponente f_j nur die $D = 2d(\mathbf{f}) + 1$ zusammenhängenden Komponenten des Argumentvektors $\mathbf{y}$ benötigt (s. Abbildung 7.5). Wird eine **blockweise Aufteilung** des ODE-Systems verwendet, d. h. das ODE-System wird in $n_B = n/B$ Blöcke unterteilt, und gilt $B \geq d(\mathbf{f})$, so werden für die Funktionsauswertungen der Komponenten im Block J nur die Komponenten der Blöcke $J-1$, J und $J+1$ des Argumentvektors gebraucht (vgl. Abbildung 7.5). Die beschränkte Zugriffsdistanz bewirkt, dass zur Berechnung des Blocks J der Argumentvektoren $\mathbf{Y}_1^{(k)}, \ldots, \mathbf{Y}_s^{(k)}$ im Korrektorschritt k nur die drei Blöcke $J-1$, J und $J+1$ der Argumentvektoren $\mathbf{Y}_1^{(k-1)}, \ldots, \mathbf{Y}_s^{(k-1)}$ aus dem Zeitschritt $k-1$ bekannt sein müssen. Dadurch eröffnet sich die Möglichkeit, Speicherplatz durch ein Überlappen der Argumentvektoren $Y^{(k)} = \left(\mathbf{Y}_1^{(k)}, \ldots, \mathbf{Y}_s^{(k)}\right)^T \in \mathbb{R}^{s,n}$ mit $k = 1, \ldots, m$ einzusparen [KR07b]. Diese Möglichkeit wurde in den Implementierungen *PipeDb1m* und *PipeDb1mt*

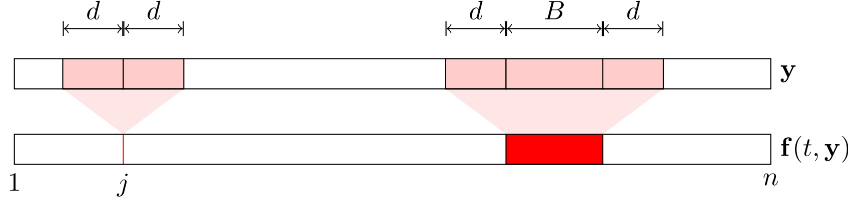


Abbildung 7.5.: Das Zugriffsmuster der Funktion **f** bei einer beschränkten Zugriffsdistanz [Kor12].

umgesetzt. Außerdem ermöglicht die beschränkte Zugriffsdistanz eine verzögerte Berechnung der Korrektorschritte gemäß dem Pipelining-Ansatz [KR07b]. Das Pipelining der Korrektorschritte wurde in den Implementierungen *ppDb1m* und *ppDb1mt* realisiert. Zur Implementierung des Pipelining-Ansatzes für die Berechnung der Korrektorschritte findet eine Schleifenvertauschung statt, bei der die Systemdimensionschleife j die äußerste Schleife wird. Dies macht die Anwendung des Loop-Tilings auf alle Schleifen, die sich im Schleifenkomplex der Schleife j befinden, möglich. Dadurch können die Arbeitsräume der Schleifen, die über die Systemgröße n iterieren, so verringert werden, dass deren Daten in den Cache passen und somit wiederverwendet werden können. Das kann sich insbesondere bei großen ODE-Systemen, deren Systemgröße n (notwendiger Speicherplatz für die Abspeicherung der Vektoren der Größe n) die Größe des Caches übersteigt, lohnen.

Die Tabelle 7.2 fasst die im Kandidatenpool enthaltenen spezialisierten Implementierungsvarianten [KR07b] zusammen. Die einzelnen Varianten werden in den nächsten Abschnitten näher beschrieben.

Tabelle 7.2.: Spezielle Implementierungsvarianten.

Implementierung	Datenstrukturen	Schleifenstruktur	Eigenschaften
PipeDb1m	$\bar{Y} \in \mathbb{R}^{s,(m-1)\cdot 2B+n}$	*k–j–i–jj–l*	ähnlich zu *PipeDb2m*, aber zur Minimierung des Speicherplatzbedarfs Überlappung der $\mathbf{Y}_l^{(k)}$ Vektoren
PipeDb1mt	$\bar{Y} \in \mathbb{R}^{s,(m-1)\cdot 2B+n}$, $\mathbf{F} \in \mathbb{R}^B$	*k–j–i–(jj)–l–jj*	ähnlich zu *PipeDb1m*, aber zusätzlich Loop-Tiling der l-Schleife
ppDb1m	$\bar{Y} \in \mathbb{R}^{s,(m-1)\cdot 2B+n}$	*j–k–i–jj–l*	basiert auf *PipeDb1m*; j- und k-Schleifen sind durch das Pipelining vertauscht
ppDb1mt	$\bar{Y} \in \mathbb{R}^{s,(m-1)\cdot 2B+n}$, $\mathbf{F} \in \mathbb{R}^B$	*j–k–i–(jj)–l–jj*	ähnlich zu *ppDb1m*, allerdings mit Loop-Tiling der l-Schleife

Überlappung der Argumentvektoren: Implementierungen PipeDb1m und PipeDb1mt

Wie schon oben erwähnt, wurde die Implementierung *PipeDb1m* mit dem Ziel entwickelt, Speicherplatz einzusparen. Die Implementierung *PipeDb1m* hat die gleiche Schleifenstruktur wie die Implementierung *PipeDb2m*, verwendet aber im Gegensatz zu *PipeDb2m* andere Datenstrukturen (s. Tab. 7.2). Allgemein gilt, dass in jedem Zeitschritt κ des Iterationsprozesses (3.27) zur Berechnung der Argumentvektoren $\mathbf{Y}_1^{(k)}, \ldots, \mathbf{Y}_s^{(k)}$ die im vorherigen Korrektorschritt $\kappa - 1$ berechnete Argumentvektoren $\mathbf{Y}_1^{(k-1)}, \ldots, \mathbf{Y}_s^{(k-1)}$ gebraucht werden. Aus diesem Grund speichern die allgemeinen Implementierungsvarianten, wie z. B. *PipeDb2m*, zwei Matrizen der Größe $s \times n$. Besitzt die Funktion $\mathbf{f}$ jedoch eine beschränkte Zugriffsdistanz, so kann der Speicherplatz zur Abspeicherung der Argumentvektoren $Y^{(k)} = \left(\mathbf{Y}_1^{(k)}, \ldots, \mathbf{Y}_s^{(k)}\right)^T$ um ca. Faktor zwei reduziert werden. Dies geschieht durch die Überlappung der m Matrizen $Y^{(k)} = \left(\mathbf{Y}_1^{(k)}, \ldots, \mathbf{Y}_s^{(k)}\right)^T \in \mathbb{R}^{s,n}$ mit $k = 1, \ldots, m$ zu einer Matrix $\bar{Y} = (\bar{y}_{l,j})$ der Größe $s \times (m-1) \cdot 2B + n$. Dabei bezeichnet die Größe B die Blockgröße, die das ODE-System in $n_B = n/B$ Blöcke unterteilt. Da die Funktionsauswertungen zur Berechnung eines Blocks J nur von den Berechnungen der Argumentvektoren der Blöcke $J-1$, J und $J+1$ abhängen, muss bei dieser Implementierungsvariante die Bedingung $B \geq d(\mathbf{f})$ gelten. Die Größe B wird in der Implementierung *PipeDb1m* auch zum Loop-Tiling der j-Schleife mit der i-Schleife verwendet.

Die Überlappung der Argumentvektoren (m Matrizen $Y^{(k)} = \left(\mathbf{Y}_1^{(k)}, \ldots, \mathbf{Y}_s^{(k)}\right)^T$ zu einer Matrix $\bar{Y} = (\bar{y}_{l,j})$) ist in der Abbildung 7.6 dargestellt. Das ODE-System wurde durch eine block-basierte Aufteilung in 10 Blöcke der Größe B unterteilt. Ein Block J speichert die Ergebnisse der Funktionsauswertungen für alle Stufen s. Man sieht, dass zur Funktionsauswertung der Komponenten im Block J der Block J und außerdem noch zwei weitere Blöcke $J+1$ und $J-1$ gebraucht werden. Aufgrund dessen werden die Ergebnisse der Funktionsauswertungen des Blocks J im Korrektorschritt k um zwei Blöcke versetzt im Block $J+2$ der Matrix $\bar{Y}$ (blaue Blöcke in der Abbildung) gespeichert. Verfolgt man dieses Abspeicherungsmuster weiter, so wird ersichtlich, dass in der Matrix $\bar{Y}$ insgesamt $(2(m-1) + n_B) \cdot s$ Blöcke gespeichert werden müssen.

Die Größe des Arbeitsraumes eines Zeitschrittes kann bei der Implementierungsvariante *PipeDb1m* durch die Formel $WS_{ts} = ((m-1) \cdot 2B + n) \cdot s + 3n$ angegeben werden. Die Matrix $\bar{Y}$ hat die Größe $((m-1) \cdot 2B + n) \cdot s$, dabei ist B die Blockgröße, die das ODE-System in $n_B = n/B$ Blöcke unterteilt. Darüber hinaus werden noch drei Vektoren $\Delta\boldsymbol{\eta}^{(m)}$, $\Delta\boldsymbol{\eta}^{(m-1)}$ und $\boldsymbol{\eta}$ der Größe n benutzt.

Bei der Implementierungsvariante *PipeDb1mt* wurde das Loop-Tiling der j-Schleife auf die l-Schleife ausgedehnt, um die räumliche Lokalität weiter zu verbessern. Bei dieser Variante kann die Größe des Arbeitsraumes eines Zeitschrittes durch die Formel $WS_{ts} = ((m-1) \cdot 2B + n) \cdot s + 3n + B$ ausgedrückt werden.

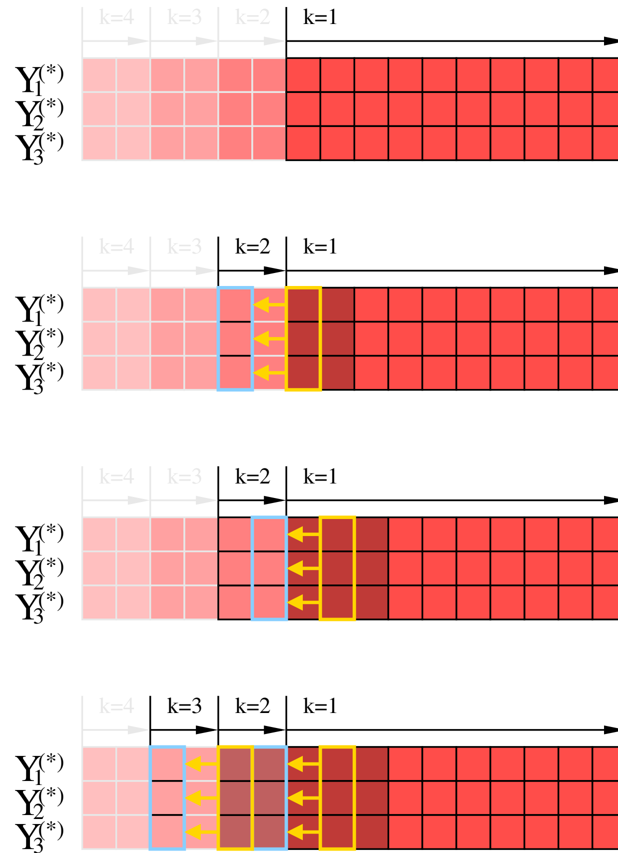


Abbildung 7.6.: Illustration der Abspeicherung von Ergebnissen der Funktionsauswertungen bei der Überlappung der Argumentvektoren für $m = 4$, $s = 3$ und $n_B = 10$ [KR07a].

Pipelining-Implementierungen: ppDb1m und ppDb1mt

Bei den Implementierungsvarianten *PipeDb1m* und *PipeDb1mt* ist der Arbeitsraum der äußeren Schleife k zwar um Faktor zwei reduziert worden, der Arbeitsraum der inneren Schleifen hängt aber immer noch von der Systemgröße n ab und hat die Größe $\Theta(snB)$. Bei ODE-Problemen mit einer beschränkten Zugriffsdistanz kann der Arbeitsraum der äußeren Schleife weiter verkleinert werden. Dafür wurde die Schleifenstruktur der Implementierung *PipeDb1m* so verändert, dass die Schleife j durch das Vertauschen mit der Schleife k zur äußersten Schleife geworden ist. Weil die Schleife j jetzt die äußerste Schleife ist, hängt der Arbeitsraum der inneren Schleifen der Implementierung *ppDb1m* somit nicht mehr von der Systemgröße n ab, sondern von der Anzahl der Korrektorschritte k und der Blockgröße B, die aber mindestens so groß gewählt sein muss, wie die Zugriffsdistanz $d(\mathbf{f})$. Innerhalb der Schleife j entstehen somit kleinere Arbeitsräume, die

	1	2	3	4	5	6	7
$(\boldsymbol{\eta}_\kappa) = Y^{(0)}$							
$Y^{(1)}$	1	2	4	7	11	16	21
$Y^{(2)}$	3	5	8	12	17	22	26
$Y^{(3)}$	6	9	13	18	23	27	30
$Y^{(4)}, \hat{\boldsymbol{\eta}}_{\kappa+1}$	10	14	19	24	28	31	33
$\boldsymbol{\eta}_{\kappa+1}$	15	20	25	29	32	34	35

Abbildung 7.7.: Berechnungsreihenfolge der Blöcke bei den Pipelining-Implementierungen *ppDb1m* und *ppDb1mt*. Illustration des Pipelinings für $m = 4$ Korrektorschritte. Die Zahlen in den Blöcken geben die Reihenfolge der Berechnungen wieder. Die grauen Blöcke werden in der Initialisierungsphase, die roten in der Sweep-Phase und die gelben Blöcke in der Finalisierungsphase der Pipeline berechnet.

eventuell im Cache gespeichert werden können. Das Vertauschen der Schleife k mit der Schleife j und das Ausnutzen einer beschränkten Zugriffsdistanz macht die verzögerte Berechnung der Korrektorschritte gemäß dem Pipelining-Ansatz möglich.

Die Abbildung 7.7 illustriert den Pipelining-Ansatz für die Berechnung der Korrektorschritte für $m = 4$. Die Blöcke in der Abbildung 7.7 entsprechen den Blöcken der Matrizen $Y^{(k)}$, die entsprechend dem Überlappungsmuster der Implementierung *PipeDb1m* gespeichert sind. Weitere Blöcke entsprechen den Approximationsvektoren $\boldsymbol{\eta}_{\kappa+1}$ und $\hat{\boldsymbol{\eta}}_{\kappa+1}$. Die Zahlen innerhalb der Blöcke geben die Reihenfolge der Berechnungen wieder. Weil die Zugriffsdistanz beschränkt ist, reichen maximal drei Blöcke des vorherigen Korrektorschrittes $k-1$, um einen Block des Korrektorschrittes k zu berechnen. In der Abbildung 7.7 sind z. B. die Blöcke 13, 18, und 23 notwendig, um den Block 24 zu berechnen. In jedem Schritt der Pipeline werden alle Blöcke einer Diagonale berechnet. Weil die Länge der Pipeline m ist, setzen wir bei den Pipeline-Implementierungen voraus, dass das ODE-System in mindestens m Blöcke aufgeteilt werden kann. Das Pipelining der Korrektorschritte setzt sich aus drei Phasen zusammen (betrachte die Abb. 7.7). Zur Vereinfachung der Beschreibung gehen wir davon aus, dass n ein Vielfaches von B ist.

- **Initialisierung der Pipeline**. Der Argumentvektor $\mathbf{Y}^{(0)}$ entspricht der im vorherigen Zeitschritt berechneten Approximation $\boldsymbol{\eta}_{\kappa+1}$ und muss somit nicht berechnet werden. Die Initialisierung startet mit der Berechnung des Blocks 1 des Argumentvektors $\mathbf{Y}^{(1)}$. Der Block 1 des Argumentvektors $\mathbf{Y}^{(1)}$ kann aus zwei Blocken 1 und 2 von $\mathbf{Y}^{(0)}$ berechnet werden. Sobald die Blöcke 1 und 2 von $\mathbf{Y}^{(1)}$ berechnet sind, kann der erste Block des Argumentvektors $\mathbf{Y}^{(2)}$ berechnet werden. Zum Berechnen des Blocks 5 des Argumentvektors $\mathbf{Y}^{(2)}$ sind die drei Blöcke 1, 2, 4 des Argumentvektors $\mathbf{Y}^{(1)}$ notwendig. Die Berechnung der Blöcke setzt sich solange fort, bis ein Block des Approximationsvektors $\boldsymbol{\eta}_{\kappa+1}$ berechnet wurde.

- **Berechnen der Blöcke in der Sweep-Phase**. Die Sweep-Phase startet mit der

Berechnung des Blocks $m+1$ des Argumentvektors $\mathbf{Y}^{(1)}$ (in der Abbildung 7.7 der Block 11) aus den Blöcken 4, 5, 6 des Argumentvektors $\mathbf{Y}^{(0)}$. In jedem Durchlauf der Schleife j über die Blöcke der Größe B mit $j = m+1, \ldots, n_B - 1$ wird dann eine komplette Diagonale mit jeweils einem neuen Block der Argumentvektoren $\mathbf{Y}^{(1)}$ bis $\mathbf{Y}^{(4)}$ und einem Block des Approximationsvektors $\boldsymbol{\eta}_{\kappa+1}$ berechnet.

- **Finalisierung der Pipeline**. Die Finalisierung beginnt, wenn keine komplette Diagonale, ausgehend vom letzten Block des Argumentvektors $\mathbf{Y}^{(0)}$, mehr aufgebaut werden kann.

Bei der Implementierungsvariante *ppDb1mt* wurde zusätzlich zum Loop-Tiling der j-Schleife Loop-Tiling auf die l-Schleife angewendet.

Die Pipelining-Varianten besitzen zwei wichtige Arbeitsräume. Der erste ist der Arbeitsraum eines Zeitschrittes. Dieser Arbeitsraum ist bei den Pipelining-Varianten genauso groß, wie bei den Implementierungsvarianten *PipeDb1m* bzw. *PipeDb1mt*, nämlich $((m-1) \cdot 2B + n) \cdot s + 3n$ bzw. $((m-1) \cdot 2B + n) \cdot s + 3n + B$. Der zweite wichtige Arbeitsraum umfasst die Menge der Vektoren, die während eines diagonalen Laufs über die Korrektorschritte k referenziert werden. Dieser Arbeitsraum entspricht einer Iterationen der über die Systemdimension laufenden Schleife j und wird als Arbeitsraum eines **Pipelining-Schrittes (PS)** bezeichnet. Die Arbeitsräume der Pipelining-Schritte unterschiedlicher Pipelining-Phasen sind ähnlich groß. In der Initialisierungs- und Finalisierungsphase werden aber weniger Blöcke berechnet. Der Arbeitsraum der Sweep-Phase ist für die Laufzeit einer Pipelining-Variante von größer Bedeutung, weil die meisten Blöcke in dieser Phase berechnet werden.

In einem Pipelining-Schritt der Sweep-Phase werden zum Berechnen der $(m-2)$ Korrektorschritte $(m-2) \cdot 3$ Blöcke der Größe sB der Matrix $\bar{Y}$ verwendet. Das Berechnen des ersten Pipelining-Schrittes erfordert zusätzlich noch einen Block, das Berechnen des Korrektorschrittes m noch drei Blöcke und das Berechnen des Vektors $\boldsymbol{\eta}_{\kappa+1}$ noch zwei Blöcke der Größe sB der Matrix $\bar{Y}$. Der Arbeitsraum eines Pipelining-Schrittes besteht folglich aus $3m$ Blöcken der Größe sB der Matrix $\bar{Y}$. Innerhalb eines Pipelining-Schrittes werden außerdem $m+1$ Blöcke der Größe B des Vektors $\boldsymbol{\eta}$ und jeweils ein Block der Größe B der Vektoren $\Delta\boldsymbol{\eta}^{(m)}$ und $\Delta\boldsymbol{\eta}^{(m-1)}$ verwendet.

Somit erhält man für die Größe des Arbeitsraumes eines Pipelining-Schrittes WS_{PS} die folgende Formel:

$$WS_{PS} = 3mBs + (m+1) \cdot B + 2B = ((3s+1) \cdot m + 3) \cdot B = \Theta(smB). \tag{7.1}$$

Aus der Formel (7.1) wird ersichtlich, dass durch die Überlappung der Argumentvektoren und die anschließende Realisierung des Pipelining-Algorithmus der Arbeitsraum der äußersten Schleife erheblich reduziert werden kann. Der Arbeitsraum der äußersten Schleife wird nicht mehr wie bei allgemeinen Implementierungsvarianten und den Varianten *PipeDb1m* und *PipeDb1mt* wesentlich durch die Systemgröße n bestimmt, sondern hängt von der Anzahl der Korrektorschritte m ab.

7.3. Laufzeitvergleich der Implementierungsvarianten aus dem Kandidatenpool

Im vorherigen Abschnitt wurde der Kandidatenpool beschrieben, der aus verschiedenen sequentiellen Implementierungsvarianten des expliziten PC-Verfahrens vom RK-Typ besteht.

Als nächstes betrachten wir anhand der beiden Abbildungen 7.8 und 7.9 die Laufzeitunterschiede zwischen den verschiedenen Implementierungsvarianten. Die Abbildung 7.8 zeigt die Laufzeiten für das BRUSS2D-Testproblem auf zwei verschiedenen Rechnersystemen. Die Beschreibung des BRUSS2D-Problems und der verwendeten Rechnersysteme ist im Anhang gegeben. Als Compiler wurde auf beiden Plattformen der gcc 4.6.4 verwendet. In der Abbildung sind die normierten Laufzeiten aufgetragen, d. h. die Laufzeit wurde durch die Anzahl der Zeitschritte und der Systemkomponenten geteilt.

Die normierte Laufzeiten sind unabhängig von der Problemgröße, daher sollte die Laufzeitfunktion bei Vernachlässigung von Speicherzugriffseffekten eigentlich eine konstante Funktion ergeben. Da aber moderne Rechnersysteme über ein hierarchisch aufgebautes Speichersystem verfügen, bestehend aus mehreren Caches mit unterschiedlichen Speicherzugriffszeiten, ändert sich die Laufzeit einer Implementierung mit der Variation der Problemgröße. Wird die Systemgröße erhöht, so wächst die Größe der Arbeitsräume eines Programms. Dadurch fallen einige Arbeitsräume aus bestimmten Cache-Stufen heraus und es kommt zu einem Anstieg der Speicherzugriffszeiten. Infolgedessen steigt in der Regel auch die normierte Laufzeit.

Betreffend der Laufzeiten der Implementierungsvarianten in der Abbildung 7.8 kann folgendes beobachtet werden:

- Der Laufzeitunterschied zwischen der langsamsten und der schnellsten Implementierungsvariante beträgt auf dem AMD DP 246 maximal 45 % und auf dem AMD Opteron 8350 maximal 30 %.

- Auf der gleichen Testplattform, aber für verschiedene Problemgrößen n, ändert sich bzgl. der Laufzeit die Reihenfolge der Implementierungsvarianten. In der Abbildung für den AMD DP 246 können drei Teilbereiche identifiziert werden, wo jeweils eine andere Implementierungsvariante die beste Performance liefert. Auf dem AMD Opteron 8350 gibt es sogar vier solcher Bereiche.

- Auf verschiedenen Plattformen, aber für die gleiche Problemgröße n, sind unterschiedliche Implementierungsvarianten jeweils die schnellsten.

Eine weitere Abbildung 7.9 demonstriert die Laufzeitunterschiede der Implementierungsvarianten in Abhängigkeit von der Systemgröße und der Stufenzahl des verwendeten ODE-Verfahrens. Es sind die Laufzeiten für das CUSP-Problem auf dem AMD Opteron 246 Rechner gezeigt. In der Abb. 7.9 (a) wurde das Radau IA (5)-Verfahren, welches drei Stufen hat, und in der Abbildung 7.9 (b) wurde das Lobatto IIIC (8)-Verfahren verwendet, dieses besitzt fünf Stufen. Aufgrund periodischer Randbedingungen

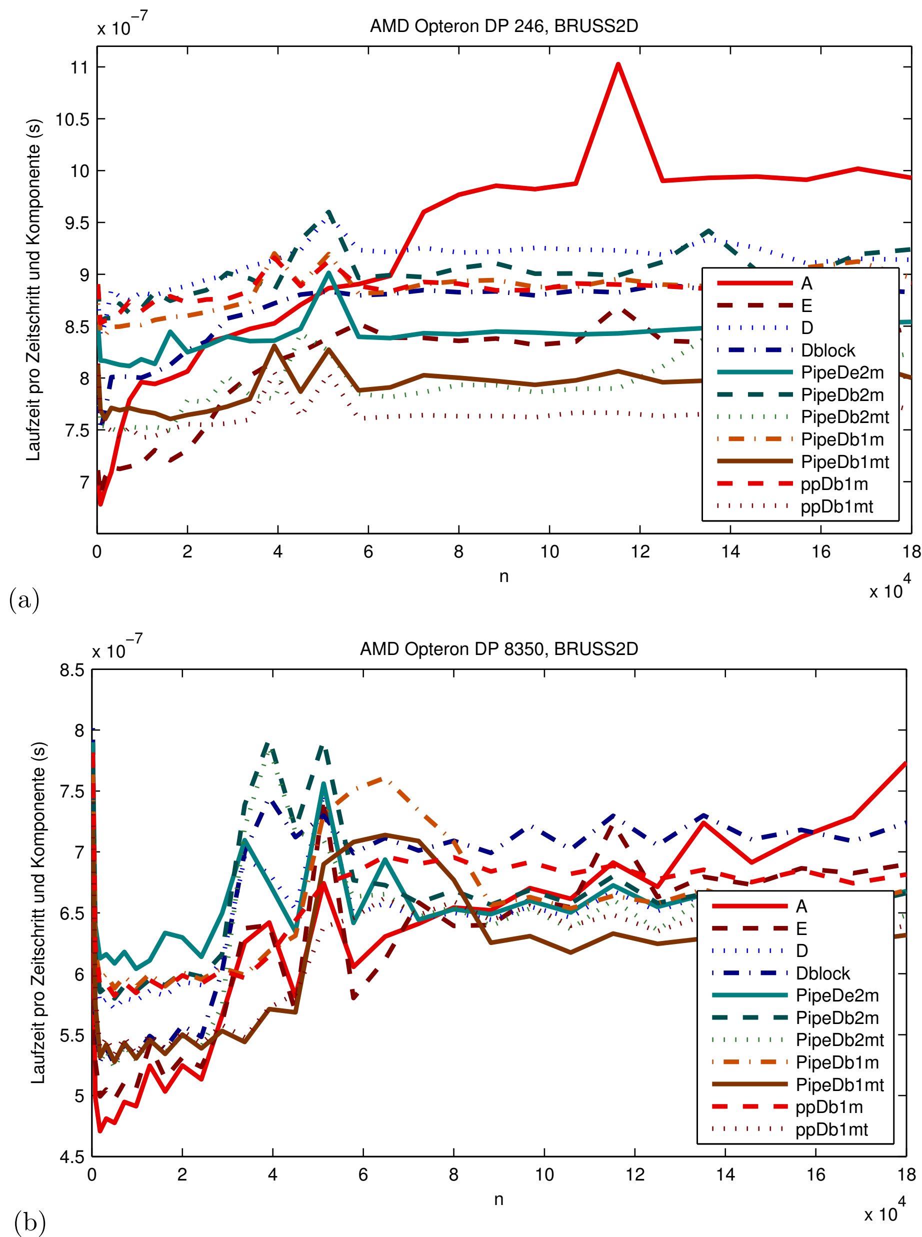


Abbildung 7.8.: Laufzeit der sequentiellen Implementierungsvarianten pro Zeitschritt und Komponente in Abhängigkeit von der Systemgröße für das BRUSS2D-Testproblem und das Radau IA (5)-Verfahren.

ist die Zugriffsdistanz beim CUSP-Problem nicht beschränkt (s. Anhang A.4). Dadurch können die spezialisierten Varianten nicht verwendet werden.

Betrachtetet man die Abbildung 7.9, so wird folgendes deutlich:

- Durch das Verwenden eines anderen ODE-Verfahrens ändert sich die Laufzeit-

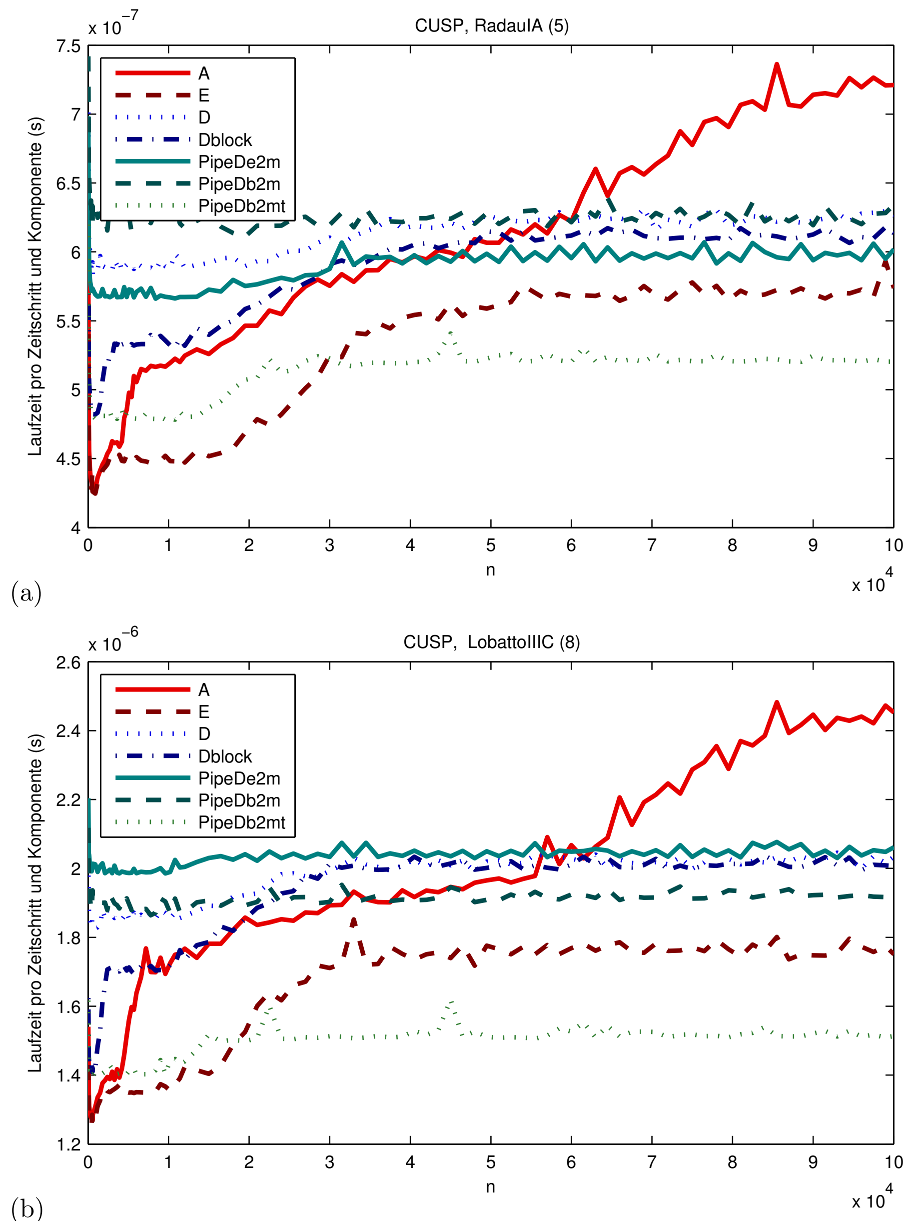


Abbildung 7.9.: Laufzeit der sequentiellen Implementierungsvarianten pro Zeitschritt und Komponente in Abhängigkeit von der Systemgröße und der Stufenzahl des verwendeten ODE-Verfahrens für das CUSP-Testproblem auf dem AMD Opteron 246 Rechnersystem. (a) Radau IA (5)-Verfahren, (b) Lobatto IIIC (8)-Verfahren.

Reihenfolge einiger Implementierungsvarianten.

- Es gibt einen Bereich von Blockgrößen, in dem für die gleiche Problemgröße n, aber für unterschiedliche ODE-Verfahren, jeweils eine andere Implementierungsvariante die geringste Laufzeit erreicht.

Allgemein kann festgehalten werden, dass die Laufzeit der Implementierungsvarianten aus dem Kandidatenpool von den Eigenschaften der Zielplattform, dem Zugriffsverhalten und der Größe des ODE-Systems, und nicht zuletzt von der Stufenzahl des verwendeten Korrektorverfahrens abhängt. Auch die Wahl des Compilers kann das Laufzeitverhalten einer Implementierungsvariante beeinflussen. Eine Implementierungsvariante, die für alle möglichen Testfälle die beste Laufzeit liefert, existiert i. d. R. nicht.

7.4. Vorüberlegungen zur Entwicklung eines Autotuning-Ansatzes

Vor der Entwicklung eines Autotuning-Ansatzes stellt sich die Frage, wie ein Autotuning speziell für die ODE-Lösungsverfahren realisiert werden kann. Vor allem muss die Wahl zwischen der Verwendung eines Offline- und eines Online-Autotuning-Konzepts getroffen werden.

Zur Gewinnung von Informationen über das Verhalten von Codevarianten auf der Zielplattform führen alle empirischen Autotuner Laufzeittests durch. Die Tests dienen der Evaluierung von Implementierungsvarianten und Parameterkonfigurationen. Insofern ist die erste Frage, die sich beim Entwurf eines Autotuning-Konzepts stellt, zu welchem Zeitpunkt die erforderlichen Tests gestartet werden sollen. Es gibt dazu zwei grundlegende Möglichkeiten: Die Informationen können entweder offline oder online gesammelt werden. Beim Offline-Autotuning finden die Laufzeittests zur Zeit der Installation oder Kompilierung eines Programms statt. Da die Eingabe des Programms erst zur Laufzeit bekannt ist, werden zur Evaluation meistens kleine synthetische Benchmarks verwendet. Werden die Tests erst zur Laufzeit gestartet, wenn die Eingabe des Programms bereits bekannt ist, so läuft das Autotuning online ab.

Wie bereits die Experimente des vorherigen Abschnittes gezeigt haben, beeinflussen nicht nur die Eigenschaften der Zielplattform, sondern auch die des zu lösenden Anfangwertproblems (Zugriffsmuster der Funktion der rechten Seite und Problemgröße) in starkem Maße die Performance von Implementierungsvarianten des betrachteten ODE-Verfahrens. Werden die Informationen über das Verhalten von Codevarianten offline gesammelt, so spiegeln diese nur die Eigenschaften der Zielplattform wieder. Man gewinnt damit keine Informationen über die Laufzeit konkreterer Probleminstanzen. Außerdem müssen die Tests für jede neue Hardwareplattform erneut gestartet werden. Da der Entwickler eines Autotuners die Benutzereingabe nicht vorhersehen kann, müssen die Offline-Tests eventuell für alle möglichen Anfangswertprobleme und unterschiedliche Systemgrößen angestoßen werden, was jedoch einen großen Zeitaufwand erfordern würde. Eine weitere Schwierigkeit für die Nutzung eines reinen Offline-Autotunings stellt die

Abhängigkeit der Programmperformance von dem verwendeten Korrektorverfahren dar. Wie bereits im vorherigen Abschnitt demonstriert, kann die Stufenanzahl des verwendeten Korrektorverfahrens die Performance von Implementierungsvarianten beeinflussen. Weil das Korrektorverfahren erst nach dem Start des ODE-Solvers bekannt ist, muss bei einem Offline-Ansatz der Autotuner die Laufzeitinformationen für alle möglichen Korrektorverfahren sammeln.

Aufgrund der genannten Schwierigkeiten scheint das Konzept des Online-Autotunings, möglicherweise kombiniert mit Offline-Tests, für ODE-Solver besser geeignet zu sein. Findet das Autotuning online statt, zum Zeitpunkt in dem das zu lösende AWP und die Problemparameter bereits bekannt sind, so kann der Programmcode nicht nur an die Eigenschaften der Zielplattform, sondern auch an die Charakteristik des zu lösenden AWPs angepasst werden.

Für ein Online-Konzept spricht auch, dass die Ausrichtung der Datenstrukturen und des binären Programmcodes im Speicher einen Einfluss auf die Programmlaufzeit ausüben könne. Eine ungünstige Anordnung der Datenstrukturen kann z. B. zu einer Steigerung der Anzahl der Cache-Fehlzugriffe und einer Verschlechterung der Programmperformance führen. Durch das Verwenden der *mmalloc()* Funktion wird der Speicher dynamisch zur Programmlaufzeit vom Betriebssystem reserviert. Dadurch kann sich die Reihenfolge der Abspeicherung von Daten im Speicher bei jedem Programmneustart ändern. Das Optimieren des Programmcodes muss daher im aktuellen Speicherkontext erfolgen. Das kann aber nur geschehen, wenn das Autotuning zur Laufzeit durchgeführt wird.

Eine Voraussetzung für die Anwendbarkeit eines Online-Autotuning-Konzepts ist die iterative Struktur einer Anwendung. Iterativ bedeutet in diesem Zusammenhang, dass der Programmcode einen Bereich enthalten soll, der immer wieder wiederholt wird. Die ODE-Verfahren sind iterativ, da sie eine zeitschrittorientierte Berechnungsstruktur besitzen. Die äußerste Schleife im Programm läuft über eine Anzahl von Zeitschritten. Innerhalb eines Zeitschrittes wird immer die gleiche Berechnungsvorschrift (s. Abb. 7.1) ausgeführt. Damit ist es möglich, das Online-Autotuning zur automatischen Optimierung von ODE-Verfahren einzusetzen.

Generell gibt es zwei Möglichkeiten, zu welchem Zeitpunkt das Online-Autotuning gestartet werden soll. Entweder der Autotuning-Prozess wird vor Beginn der Berechnung gestartet, sobald die Eingabe bekannt ist. Im Kontext der ODE-Solver bedeutet dies, dass der Autotuning-Prozess vor dem ersten Zeitschritt abgeschlossen sein muss. Oder das Autotuning wird während der Berechnung durchgeführt, bei ODE-Solvern als Teil der zeitschrittorientierten Berechnungsvorschrift. Bei der ersten Option muss zusätzliche Zeit vor Beginn der Berechnung zum Ermitteln der schnellsten Implementierungsvariante investiert werden. Die schnellste Implementierungsvariante steht vor dem Beginn der Berechnung fest und alle Zeitschritte können mit der schnellsten Implementierungsvariante ausgeführt werden. Wird das Autotuning als Teil der Berechnungsvorschrift ausgeführt, so müssen mehrere Zeitschritte zur Evaluierung der Implementierungsvarianten und Programmparameter herangezogen werden. Die Laufzeit geht zwar durch das Ausführen langsamer Implementierungen und Parameter verloren, dafür tragen die

Berechnungen dieser Zeitschritte zur Lösung bei.

Infolge dieser Überlegungen wurde die Entscheidung getroffen, das Online-Konzept zur automatischen Anpassung von ODE-Verfahren an das zu lösende AWP und die Zielarchitektur zu nutzen und dieses durch Offline-Techniken zu ergänzen. Das Autotuning soll während der Berechnung durchgeführt werden. Im nächsten Abschnitt wird die Grundstruktur des Online-Autotuning-Algorithmus zur automatische Selektion der schnellsten sequentiellen Implementierungsvariante expliziter PC-Verfahren vom RK-Typ vorgestellt.

7.5. Der Autotuning-Grundalgorithmus

Die im Kandidatenpool enthaltenen Implementierungsvarianten unterscheiden sich zwar in der Schleifenreihenfolge und den Datenstrukturen, besitzen aber die gleichen numerischen Eigenschaften. Deshalb kann zwischen der Berechnung von Zeitschritten ein Wechsel von Implementierungsvarianten stattfinden. Der Grundalgorithmus des Autotunings nutzt die zeitschrittorientierte Berechnungsstruktur von ODE-Verfahren aus, um zur Laufzeit die schnellste Implementierungsvariante auf der Zielplattform auszuwählen. Der Grundalgorithmus besteht aus drei Phasen:

- **Vorauswahlphase**. Die Vorauswahlphase findet vor Beginn der Berechnung statt. In dieser Phase werden die Implementierungsvarianten aus den Kandidatenpool entfernt, die auf das zu lösende ODE-Problem nicht anwendbar sind. Die allgemeine Implementierungsvarianten können für alle ODE-Probleme eingesetzt werden, während die spezialisierten Implementierungsvarianten nur auf ODE-Probleme mit beschränkter Zugriffsdistanz anwendbar sind.

- **Autotuningphase**. Die Autotuningphase findet während der ersten Zeitschritte des Integrationsprozesses statt. In dieser Phase werden nacheinander alle im Kandidatenpool verbliebenen Implementierungsvarianten zur Berechnung je eines Zeitschrittes verwendet. Die Laufzeit jedes Zeitschrittes wird gemessen und am Ende der Autotuningphase wird die schnellste Implementierungsvariante anhand der erfassten Laufzeiten bestimmt.

- **Berechnungsphase**. Die restlichen Zeitschritte werden mit der Implementierungsvariante ausgeführt, die in der Autotuningphase als die schnellste ausgewählt wurde.

Der Pseudocode des Autotuning-Grundalgorithmus ist in der Abbildung 7.10 gegeben. Der Algorithmus kombiniert eine Offline- mit einer Online-Strategie. In der Vorauswahlphase arbeitet der Algorithmus offline. Das heißt, er trifft eine Vorauswahl der Implementierungsvarianten noch bevor der Integrationsprozess gestartet wird. Alle anderen Phasen des Algorithmus finden online statt, d. h. während des Lösens eines Anfangswertproblems.

```
 1: Sei P die zu lösende Probleminstanz;
 2: Sei C die Kandidatenmenge der Implementierungsvarianten;

 3: C_1 ← pre-select (C, P);                          // Vorauswahl von Implementierungsvarianten

 4: h ← h_init;                                       // Wähle die Anfangsschrittweite
 5: t ← t_0;                                          // Setze t auf Anfang des Integrationsintervalls
 6:
 7: wähle eine (beliebige) Implementierung aus C_1;
 8: while (t < t_e)                                   // Durchlaufe das Integrationsintervall
 9: {
10:     if (nicht alle Implementierungen in C_1 evaluiert)
11:     {
12:         führe den Zeitschritt mit der gewählten Implementierung aus;
13:         if (der zweite Zeitschritt von zwei aufeinander folgenden akzeptierten Zeitschritten)
14:         {
15:             T ← Zeit zur Berechnung des aktuellen Zeitschrittes;
16:             if (T < T_best)                       // Aktuelle Implementierung ist die schnellste bis jetzt
17:             {
18:                 beste Implementierung ← aktuelle Implementierung;
19:                 T_best ← T;
20:             }
21:             wähle eine Implementierung aus C_1, die noch nicht evaluiert wurde;
22:         }
23:     }
24:     else                                          // Alle Implementierungen wurden bereits evaluiert
25:         führe den Zeitschritt mit der Implementierung aus, die als beste ermittelt wurde;
26:     führe Schrittweitenkontrolle aus und aktualisiere t, wenn der lokale Fehler klein genug;
27: }
```

Abbildung 7.10.: Grundalgorithmus des Online-Autotunings.

7.6. Erfassen von Laufzeiten einzelner Implementierungsvarianten

Der Autotuning-Algorithmus vergleicht die verschiedenen Implementierungsvarianten anhand der gemessenen Laufzeit pro Zeitschritt. Deshalb ist es besonders wichtig, die Laufzeit eines Zeitschrittes mit der jeweiligen Implementierungsvariante korrekt zu erfassen. Dafür müssen zwei Punkte beachtet werden. Erstens, die Messung der Laufzeit in jedem Zeitschritt soll ausgehend von einem reproduzierbaren Zustand des Rechnersystems und insbesondere der Cache-Hierarchie stattfinden. Da beim Start eines Programms die Daten erst in die Cache-Hierarchie geladen werden müssen, kann in einem begrenzten Zeitraum zu Beginn der Ausführung eine hohe Anzahl von Cache-Fehlzugriffen auftreten. Sobald aber die meisten Programmdaten im Cache liegen, können die Datenzugriffe größtenteils aus dem Cache bedient werden. Dadurch geht die Anzahl der Cache-Fehlzugriffe

zurück und die Programmlaufzeit sinkt. Dieses Problem wird in der Literatur als **Problem des kalten Startes (engl. cold-start problem)** [EF78] bezeichnet. Beim Entwurf des Autotuning-Algorithmus muss dieses berücksichtigt werden, ansonsten läuft man Gefahr, die Laufzeit eines Zeitschrittes und folglich einer Implementierungsvariante als zu hoch einzuschätzen.

Zweitens sind für die genaue Zeitmessung eines Zeitschrittes **hochauflösende Zeit-Uhren (engl. High Resolution Timer)** notwendig. Zum Beispiel ist die im UNIX implementierte Zeitfunktionen *gettimeofday()* für die Zeiterfassung ungeeignet, da die zeitliche Auflösung dieser Funktion maximal im Mikrosekundenbereich liegt. Die Zeit eines Zeitschrittes oder der Laufzeitunterschied zwischen verschiedenen Implementierungen kann jedoch kleiner als der maximal messbare Zeitbereich dieser Zeitfunktion sein. Eine Möglichkeit, die Zeit mittels hochauflösender Uhren zu erfassen, bieten die POSIX-Echtzeiterweiterungen mit den Zeit-Uhren CLOCK_REALTIME oder CLOCK_MONOTONIC. Die Zeit-Uhren können durch die POSIX-Funktion *clock_gettime()* ausgelesen werden. Die CLOCK_REALTIME repräsentiert die aktuelle Zeit und kann, falls sich die Systemuhr ändert, z. B. bei einem Versetzen des Systems in Tiefschlaf, Zeitsprünge nach vorne oder nach hinten vollziehen [QM12]. Die CLOCK_MONOTONIC Uhr läuft immer vorwärts und repräsentiert die Zeit seit dem letzten Systemstart [QM12]. Da CLOCK_MONOTONIC auch bei Änderungen der Systemzeit weiter läuft, wird sie bevorzugt für die Zeitmessung eines Zeitintervalls verwendet. Beide Zeit-Uhren reagieren aber auf Änderung der Taktung, beispielsweise durch das **Netzwerk-Zeit-Protokoll (engl. Network Time Protocol (NTP)** [QM12]. Die Auflösung der POSIX-Zeituhren ist vom Betriebssystem-Kernel und der Hardware abhängig. Bei der aktuellen Hardware liegt die Auflösung im Mikro- bis Nanosekundenbereich.

Eine Alternative zum Verwenden der POSIX-Zeituhren bietet das Auslesen von Hardware-Taktzählern, über die die meisten modernen Rechnersysteme verfügen. Die Hardware-Taktzähler sind prozessorspezifische Register, die mit jedem Prozessortakt automatisch inkrementiert werden. Das Verwenden von Inline-Assembler-Code zum Auslesen von Hardware-Taktzählern kann im Vergleich zum Auslesen der POSIX-Zeituhr zu einem geringen Zeitoverhead führen. Die Schwierigkeit beim Auslesen von Hardware-Taktzählern besteht darin, dass der Zugriff auf Register je nach Architektur unterschiedlich implementiert ist. Außer den Registern zum Zählen von CPU-Zyklen verfügen die modernen Prozessoren über weitere prozessorspezifischer Register, auch **Hardware-Performance-Zähler (engl. Hardware Performance Counter)** genannt, zum Aufzeichnen von weiteren Ereignissen, wie beispielsweise der Anzahl von Cache-Misses oder von TLB-Misses. Jeder Prozess im System besitzt seine eigenen Hardware-Performance-Zähler. Ein einfaches Auslesen der Hardware-Performance-Zähler ist mit Hilfe spezieller Bibliotheken, z. B. PAPI (Performance Application Programming Interface)[pap14], möglich. Die PAPI-Bibliothek bietet dem Benutzer zwei Schnittstellen an: eine High-Level- und eine Low-Level-API (Application Programming Interface). Die High-Level-API ist einfacher zu bedienen und bietet die Möglichkeit zum Starten, Stoppen und Auslesen der Zähler für eine vordefinierte Liste von PAPI-Ereignissen. Im Gegensatz zu High-Level-Schnittstelle bietet die Low-Level-Schnittstelle mehr Funktionalität. Sie

erlaubt nicht nur das Messen vordefinierter PAPI-Ereignisse, sondern auch prozessorspezifischer Ereignisse. Da die meisten Prozessoren nur über wenige Zählerregister verfügen, kan PAPI ebenfalls nur wenige Ereignisse gleichzeitig messen.

Der implementierte Autotuner verfügt über drei Methoden zum Erfassen der Laufzeit:

- Erfassen der Laufzeit durch den Aufruf der POSIX-Funktion *clock_gettime()* mit dem Zeitgeber CLOCK_MONOTONIC.
- Zugriff auf Hardware-Taktzähler durch Inline-Assembler-Code, implementiert im KOALA-Framework [Hof09].
- Messen der Laufzeit mit High-Level-Zeitfunktionen der PAPI-Bibliothek.

Standardmäßig misst der Autotuner die Laufzeit mit Hilfe der High-Level-PAPI-Zeitfunktionen. Ist die PAPI-Bibliothek auf dem ausführenden Rechnersystem nicht installiert, so kann der Benutzer zwischen zwei anderen Methoden der Zeitmessung wählen: POSIX-Funktionen oder Auslesen der Hardware-Taktzähler mittels Funktionen des KOALA-Frameworks.

Das Problem des kalten Startens lässt sich durch das **Aufwärmen des Caches (engl. cache warm up)** vermeiden. Darunter versteht man das anfängliche Laden einer Reihe von Daten in den Cache. Um den Cache so reproduzierbar wie möglich aufzuwärmen, unterscheidet der Autotuning-Algorithmus zwischen akzeptierten und verworfenen Zeitschritten. Wird ein Zeitschritt verworfen, so muss der alte Approximationsvektor wiederhergestellt werden. Dazu muss der Vektor $\boldsymbol{\eta}_{\text{old}}$ in den Vektor $\boldsymbol{\eta}$ kopiert werden (s. Rahmenprogramm in der Abbildung 7.1). Weil im Laufe des Integrationsprozesses nur eine kleine Anzahl der Zeitschritte verworfen wird, misst der Autotuner die Laufzeit von Implementierungsvarianten anhand der akzeptierten Zeitschritte. Der erste akzeptierte Zeitschritt wird genutzt, um den Cache mit der entsprechenden Implementierung aufzuwärmen. Durch das Ausführen eines Zeitschrittes werden die Daten dieser Implementierung in den Cache geladen und können im nächsten Zeitschritt wiederverwendet werden. Die Laufzeit einer Implementierungsvariante wird erst für den zweiten der akzeptierten Zeitschritte gemessen, der unmittelbar auf den ersten akzeptierten Zeitschritt folgt. Diese Zeit dient als Grundlage für den Vergleich unterschiedlicher Implementierungsvarianten.

7.7. Experimentelle Evaluierung des Autotuning-Grundalgorithmus

In diesem Abschnitt wird anhand der Laufzeitexperimente auf verschiedenen Zielsystemen und für verschiedene ODE-Probleme die Anwendbarkeit des Autotuning-Grundalgorithmus auf die Klasse der PC-Verfahren vom RK-Typ gezeigt. Die Beschreibung der verwendeten Testprobleme und Rechnersysteme ist im Anhang gegeben.

Die Abbildung 7.11 präsentiert einen detaillierten Vergleich der Laufzeiten der im Kandidatenpool enthaltenen, nicht-adaptiven Implementierungsvarianten und der Autotuning-Implementierung. Bei den Laufzeitexperimenten wurden das BRUSS2D-Problem

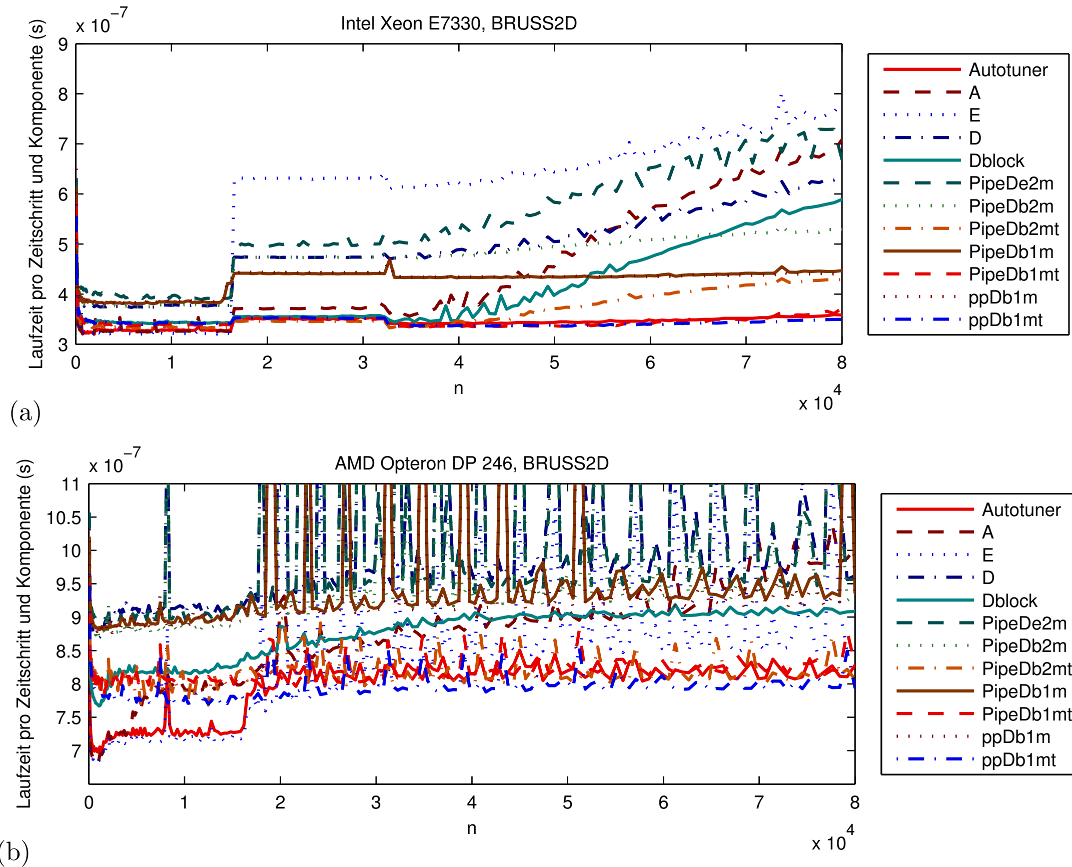


Abbildung 7.11.: Vergleich der normierten Laufzeiten nicht-adaptiver Implementierungsvarianten und der Autotuning-Implementierung für das BRUSS2D Testproblem und das Radau IA (5)-Verfahren. Die Laufzeiten sind in Abhängigkeit der Systemgröße n aufgetragen. (a) Intel Xeon E7330, (b) AMD Opteron DP 246.

und das Radau IA (5)-Verfahren verwendet. In der Abbildung sind normierte Laufzeiten (vgl. Abschnitt 7.3) in Abhängigkeit von der Systemgröße n aufgetragen. Um die Zeit zur Durchführung einer Experimentenreihe mit allen Implementierungsvarianten gering zu halten, wurde die Gesamtzahl der auszuführenden Zeitschritte eingeschränkt. Es wurde aber sichergestellt, dass die Autotuning-Implementierung mindestens dreimal so viele Schritte ausführt wie in der Autotuningphase, und dass die Ausführungszeit eines Programms lang genug ist, ca. 30 s, um diese zuverlässig messen zu können. Als Ergebnis führt die Autotuning-Implementierung in den Experimenten zwischen 60 und 850 Zeitschritte durch.

Die Abbildung 7.11 (a) zeigt die Ergebnisse auf dem Intel Xeon E7330 System. Auf diesem System bleibt die Laufzeit für Systemgrößen kleiner 16384 für alle Implementierungsvarianten nahezu konstant. Erreicht die Systemgröße n den Wert 16384, so kann

ein steiler Anstieg der Laufzeit für alle Implementierungsvarianten beobachtet werden. Die Ursache für diesen Anstieg ist der Wechsel der von der *glibc* Funktion *malloc()* verwendeten Strategie zum dynamischen Allokieren von Speicher. Normalerweise stellt die *malloc()* Funktion Speicher auf dem Heap bereit und passt die Größe des Heaps mittels der Funktion *sbrk()* an. Sollen jedoch größere Blöcke allokiert werden, deren Größe MMAP_THRESHOLD Bytes übersteigt, so findet ein Strategiewechsel statt und der Speicher wird mittels der *mmap()* Funktion allokiert. Der in Linux vordefinierte Wert für die MMAP_THRESHOLD Konstante ist 128 KB, dies entspricht den 16384 Double Werten. Der Wert MMAP_THRESHOLD kann mittels der Funktion *mallopt()* verändert werden.

Der Intel Xeon E7330 Rechner besitzt eine zweistufige Cache-Hierarchie. Der größte Cache ist der L2-Cache mit einer Größe von 3 MB. Wie bereits im Abschnitt 7.2.2 erwähnt, entspricht der Gesamtspeicherplatzbedarf einer Implementierungsvariante der Größe des Arbeitsraumes eines Zeitschrittes. Der von den allgemeinen Implementierungsvarianten benötigte Speicherplatz ist von der Anzahl der Stufen s und der Systemgröße n abhängig. Bei Loop-Tiling-Varianten hängt der Speicherplatzbedarf zusätzlich von der Blockgröße B ab. Der von den Implementierungsvarianten A und E benötigte Speicherplatz ist durch $(2s + 4) \cdot n$, der von den Implementierungsvarianten D, *PipeDe2m*, *PipeDb2m* durch $(2s+3)\cdot n$ gegeben (s. Abschnitt 7.2.2). Bei den Loop-Tiling-Varianten *Dblock*, *PipeDb2mt* ist der Speicherplatzbedarf durch $(2s + 3) \cdot n + B$ gegeben.

Der Arbeitsraum eines Zeitschrittes der Implementierungsvarianten A und E passt bis zu einer Systemgröße $n \lesssim 3.9 \cdot 10^4$, der von der Implementierung D abgeleiteten Implementierungsvarianten bis $n \lesssim 4.3 \cdot 10^4$ in den L2-Cache. Steigt die Systemgröße n über diese Werte hinaus, so fallen nach und nach Teile des Arbeitsraumes eines Zeitschrittes aus dem L2-Cache heraus. Dies äußert sich vor allem in einem Anstieg der Laufzeit.

Die spezialisierten Implementierungsvarianten *PipeDb1m*, *PipeDb1mt*, *ppDb1m*, *ppDb1mt* nutzen eine beschränkte Zugriffsdistanz der rechten Seite $\mathbf{f}$ aus, um Speicherplatz für die Berechnung der Korrektorschritte einzusparen. Statt zwei Matrizen der Größe $s \times n$ brauchen die spezialisierten Implementierungsvarianten nur eine Matrix der Größe $s \times (m - 1) \cdot 2B + n$, um die Argumentvektoren $Y^{(k)} = \left(\mathbf{Y}_1^{(k)}, \ldots, \mathbf{Y}_s^{(k)}\right)^T$ im Korrektorschritt k zu speichern. Dabei ist B die Blockgröße, die das ODE-System in $n_B = n/B$ Blöcke unterteilt. Weil der Arbeitsraum eines Zeitschrittes die Größe $s \cdot ((m - 1) \cdot 2B + n) + 3n$ hat (vgl. Abschnitt 7.2.3), überschreitet dieser die Größe des L2-Caches erst ab einer Systemgröße $n \gtrsim 6.4 \cdot 10^4$. Für kleinere Werte von n bleibt die normierte Laufzeit nahezu konstant, für größere Werte steigt die Laufzeit der Implementierungsvarianten *PipeDb1m*, *PipeDb1mt* an.

Der Arbeitsraum eines Zeitschrittes der Pipelining-Varianten *ppDb1m* und *ppDb1mt* ist gleich groß wie der von Implementierungen *PipeDb1m* bzw. *PipeDb1mt*. Da aber die Implementierungsvarianten *ppDb1m* und *ppDb1mt* das Pipelining-Prinzip zur verzögerten Berechnung der Korrektorschritte nutzen, ist bei diesen Varianten der wichtigste Arbeitsraum der des Pipeline-Schrittes. Der Arbeitsraum eines Pipeline-Schrittes (vgl. Abschnitt 7.2.3) ist viel kleiner als der Arbeitsraum eines Zeitschrittes und passt für Systemgrößen $n \lesssim 2.7 \cdot 10^7$ in den L2-Cache. Aus diesem Grund steigt die normierte

Laufzeit der Pipelining-Varianten nur geringfügig an, wenn die Systemgröße den Wert $n \gtrsim 6.4 \cdot 10^4$ übersteigt.

Vergleicht man die unterschiedlichen Implementierungsvarianten untereinander, so sieht man, dass die Laufzeit-Reihenfolge der Implementierungen für verschiedene Systemgrößen variiert. In Abbildung 7.11 (a) können drei wesentliche Teilbereiche identifiziert werden, in denen jeweils unterschiedliche Varianten die beste Performance erreichen. Für kleine Systemgrößen $n < 16384$ liefert die Variante E die schnellste Laufzeit. Für $16384 \leq n \lesssim 3.8 \cdot 10^4$ ist die beste Implementierung *PipeDb2mt*. Für noch größere Systemgrößen ist die Implementierung *ppDb1mt* die schnellste.

Beim Vergleich der nicht-adaptiven Implementierungsvarianten mit der Autotuning-Implementierung erreicht die Autotuning-Implementierung auf allen drei Teilbereichen annähernd die gleiche Performance wie die schnellste nicht-adaptive Implementierung. Der zusätzliche Zeitbedarf, der benötigt wird, die schnellste Implementierungsvariante dynamisch auszuwählen, liegt unter 3 %. Der Autotuning-Overhead ist somit sehr klein im Vergleich zum Performanceverlust von mehr als 200 %, der entstehen würde, wenn die schlechteste Implementierungsvariante (hier die Implementierung E für $n = 8.0 \cdot 10^4$) gewählt werden würde. In der Abbildung wird auch ersichtlich, dass keine Implementierungsvariante die schnellste für alle Systemgrößen ist. In diesem Kontext ist es besonders interessant, dass die Implementierung E für $n < 16384$ die schnellste Implementierung ist, für $n \geq 16384$ aber die langsamste. Selbst wenn die Implementierungsvariante *ppDb1mt* für alle Systemgrößen gewählt wird, weil sie die schnellste Variante für einen großen Bereich der n-Werte ist, so wird für Systemgrößen $n < 16384$ der Performanceverlust im Vergleich zu Implementierung E größer als 4.9 % und im Vergleich zu Autotuning-Implementierung größer als 3 % sein.

Auf dem AMD Opteron DP 246 System in Abbildung 7.11 (b) kann ein ähnliches Verhalten der Implementierungsvarianten wie auf dem Intel Xeon E7330 System beobachtet werden. Der sichtbare Unterschied besteht jedoch in dem Vorhandensein einer großen Anzahl von Laufzeitspitzen. Die hohen Laufzeitspitzen treten bei den Implementierungsvarianten auf, bei denen in inneren Schleifen über die Stufen iteriert wird. Das sind die Implementierungsvarianten *E*, *D*, *PipeDe2m*, *PipeDb2m*, *PipeDb1m*, *ppDb1m*, aber nicht die Varianten *A*, *Dblock*, *PipeDb2mt*, *PipeDb1mt* und *ppDb1mt*. Die Höhe der Laufzeitspitzen steigt für Systemgrößen $n \geq 16384$ an. Durchgeführte Experimente auf verschiedenen Rechnersystemen und mit unterschiedlichen RK-Methoden haben gezeigt, dass solche hohen Laufzeitspitzen nur auf Opteron-Prozessoren und nicht auf Xeon- und Itanium-Prozessoren auftreten. Man kann deshalb davon ausgehen, dass die Iterationen über die Stufen in innersten Schleifen zu Konfliktfehlzugriffen im L1-Cache der Opteron-Prozessoren führen. Der L1-Cache ist bei Opteron-Prozessoren im Gegensatz zu anderen Prozessoren nur 2-fach assoziativ. Generell gilt, je kleiner die Assoziativität des Caches, desto größer ist die Wahrscheinlichkeit eines Cache-Misses bei gleich bleibender Cachegröße.

Ähnlich wie auf dem Intel Xeon E7330 System, können auf dem AMD Opteron DP 246 System zwei Teilbereiche identifiziert werden, in denen jeweils unterschiedliche Implementierungsvarianten die beste Performance erreichen. Implementierung E ist die

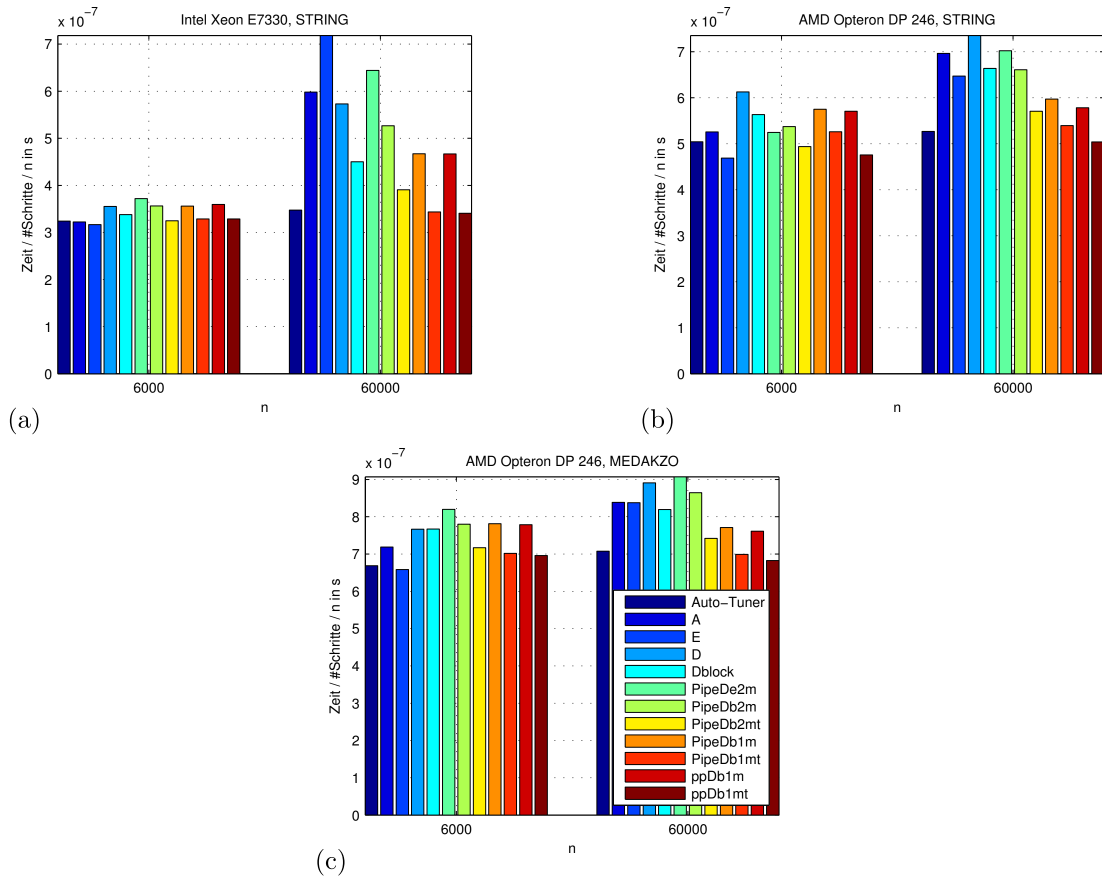
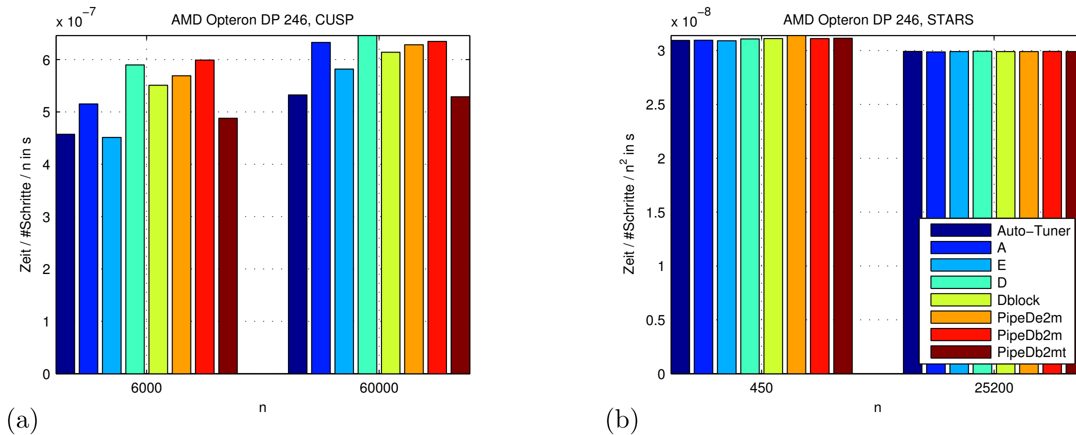


Abbildung 7.12.: Laufzeit der Implementierungen für ODE-Probleme mit beschränkter Zugriffsdistanz auf dem AMD Opteron DP 246 und dem Intel Xeon E7330 Rechnern.

schnellste für $n = 2.3 \cdot 10^4$. Für größere Werte von n zeigt die Implementierungsvariante *ppDb1mt* das beste Laufzeitverhalten. Auch auf dem AMD-Opteron-System wählt die Autotuning-Strategie für jede Systemgröße die beste Implementierungsvariante aus und das mit geringem Zeitaufwand (kleiner als 3.2 %). Würde man auf diesem System für alle Systemgrößen n die Implementierungsvariante E wählen, so würde der Performanceverlust zu der Autotuning-Implementierung 10.2 % betragen. Bei der Wahl der Implementierungsvariante E muss man im Vergleich zu der Autotuning-Implementierung mit einem Performanceverlust von 7.3 % rechnen.

Die Abbildungen 7.12 und 7.13 demonstrieren, dass der Autotuning-Ansatz auch bei anderen Testproblemen als BRUSS2D erfolgreich funktioniert. In der Abbildung 7.12 wird die Performance der Autotuning-Implementierung mit den nicht adaptiven Implementierungsvarianten anhand von Testproblemen mit beschränkter Zugriffsdistanz verglichen. Die Balkendiagramme 7.12 (a) und (b) zeigen die normierten Laufzeiten für das STRING-Problem auf dem Opteron und dem Xeon-Rechner. Auf beiden Systemen



(a) (b)

Abbildung 7.13.: Laufzeit der Implementierungen für ODE-Probleme mit unbeschränkter Zugriffsdistanz auf AMD Opteron DP 246.

ist für kleine Systemgrößen die Variante E die schnellste und für große Systemgrößen die Variante *ppDb1mt*. Außerdem ist die Variante E auf beiden Rechnern für große Systemgrößen die langsamste Implementierung. Auf dem Xeon-Rechner, für die Systemgröße $n = 6000$, ist die Autotuning-Implementierung nur 2 % langsamer als die schnellste Implementierung E und 14 % schneller als die langsamste Implementierung *PipeDe2m*. Für $n = 60000$ ist die Autotuning-Implementierung doppelt so schnell wie die langsamste Implementierung E und nur 1 % langsamer als die schnellste Implementierung *ppDb1mt*. Auf dem Opteron-Rechner liefert die Autotuning-Implementierung für das STRING (Abb. 7.12 (b)) bzw. das MEDAKZO-Problem (Abb. 7.12 (c)) ähnlich gute Ergebnisse wie auf dem Xeon-Rechner.

Da die Testprobleme CUSP und STARS eine unbeschränkte Zugriffsdistanz besitzen, können zur ihrer Lösung nur die allgemeine Implementierungsvarianten verwendet werden. Die Abbildung 7.13 zeigt den Laufzeitvergleich zwischen allgemeinen Implementierungsvarainten und der Autotuning-Implementierung auf dem Opteron System. Für das CUSP-Problem und die Systemgröße $n = 6000$ (Abb. 7.13 (a)) ist die Autotuning-Implementierung 1 % langsamer als die schnellste Implementierung E und 31 % schneller als die langsamste Implementierung *PipeDb2m*. Für die Systemgröße $n = 60000$ ist die Autotuning-Implementierung 1 % langsamer als die schnellste Implementierung E und 21 % schneller als die langsamste Implementierung D.

Beim STARS-Problem (Abb. 7.13 (b)) sieht man kaum Unterschiede in den normierten Laufzeiten der Implementierungsvarianten. Der Grund dafür ist, dass das STARS-Problem **dichtbesetzt** ist. Von einem dichtbesetzten ODE-System spricht man, wenn zum Berechnen einer Funktionskomponente f_j alle oder sehr viele Elemente des Argumentvektors $\mathbf{y}$ gebraucht werden. Dagegen werden bei **dünnbesetzten** ODE-Systemen zum Berechnen einer Funktionskomponente f_j nur wenige Elemente des Argumentvektors $\mathbf{y}$ im Vergleich zur Systemgröße n benötigt. Wie schon im Abschnitt 7.2.3 beschrieben, sind Probleme mit beschränkten Zugriffsdistanz ein Spezialfall dünnbesetzter

ODE-Systeme. Beim STARS-Problem wird für die Berechnung der Funktionskomponente f_j mit $j = 1, 3, 5, \ldots, n-1$ auf $n/2$ Elemente des Argumentvektors $\mathbf{y}$ zugegriffen. Das Lokalitätsverhalten und die Laufzeit der Implementierungsvarianten wird deshalb hauptsächlich von den Speicherzugriffen der Funktionsauswertung und deren Kosten bestimmt. Die Schleifenstruktur einer Implementierungsvariante trägt dagegen nur einen kleinen Anteil zur Gesamtlaufzeit bei.

7.8. Zusammenfassung

In diesem Kapitel ist ein Grundalgorithmus für die automatische Selektion sequentieller Implementierungsvarianten des betrachteten PC-Verfahrens vorgestellt worden. Die Laufzeitexperimente für verschiedene ODE-Probleme und unterschiedliche Zielarchitekturen haben die Anwendbarkeit des Grundalgorithmus bestätigt. In den durchgeführten Experimenten war der Autotuning-Grundalgorithmus in der Lage, entweder die schnellste Implementierungsvariante oder nahezu die schnellste Implementierungsvariante zu bestimmen. Somit ist die Auswahl einer schnellen Implementierungsvariante zur Laufzeit möglich.

Der Zeitaufwand zum Ermitteln der besten Variante ist von der Gesamtanzahl der Zeitschritte zur Lösung des Testproblems und der Anzahl der Implementierungsvarianten im Kandidatenpool abhängig. Die Experimente in diesem Kapitel haben jedoch gezeigt, dass der Zeitaufwand zur Auswahl einer schnellen Implementierungsvariante gering ist. Das gilt selbst für die in den Experimenten verwendete relativ kleine Anzahl von Zeitschritten zwischen 60 und 850. Bei echten numerischen Simulationen kann die Lösung eines Anfangswertproblems mehrere Hunderttausende bis Millionen Zeitschritte erfordern. Dies würde sogar den Einsatz eines viel größeren Kandidatenpools rechtfertigen.

8. Automatische Auswahl von Blockgrößen für sequentielle Loop-Tiling-Varianten

Im letzten Kapitel wurde ein Autotuning-Algorithmus zur automatischen Auswahl sequentieller Varianten des betrachteten PC-Verfahrens vom RK-Typ vorgestellt. Die Anwendbarkeit des Algorithmus wurde auf mehreren Testplattformen und für verschiedene ODE-Probleme gezeigt. Der Algorithmus bestimmt anhand der Eigenschaften des zu lösenden ODE-Systems dessen Zugehörigkeit zu einer Klasse von Systemen mit beschränkter oder unbeschränkter Zugriffsdistanz. Zu jeder Klasse stellt der Algorithmus eine Menge von Implementierungsvarianten bereit. Die beste Implementierungsvariante wird anhand eines Vergleichs der Laufzeiten während der ersten Zeitschritte des Integrationsprozesses bestimmt.

Der vorgestellte Algorithmus wählt die beste Implementierungsvariante aus, ohne dabei die Programmparameter, wie z. B. die Blockgröße, zu optimieren. Die Optimierung von Programmparametern ist jedoch unabdingbar zum Erreichen einer hohen Performance. Bei Loop-Tiling-Varianten kann die räumliche und zeitliche Lokalität der Speicherzugriffe durch die Wahl einer geeigneten Blockgröße gesteigert werden. Die Blockgröße sollte nicht zu klein gewählt werden, ansonsten wird der Cache nicht voll ausgenutzt und die Zeit zum Durchführen der Schleifenkontrolle wird den Laufzeitgewinn des Loop-Tilings aufzehren. Anderseits soll die Blockgröße auch nicht zu groß gewählt werden, sonst passen unter Umständen nicht alle Daten eines Blocks in den Cache. Dies kann zu einer Vergrößerung der Anzahl von Cache-Fehlzugriffen und folglich zu einer Verschlechterung der Programmperformance führen.

Das Problem der Auswahl geeigneter Blockgrößen ist anspruchsvoll. Bei den Loop-Tiling-Varianten des betrachten RK-Verfahrens ist die Effizienz einer Blockgröße von der Systemdimension, den Eigenschaften des Hardwaresystems, dem Speicherzugriffsverhalten einer Implementierungsvariante und nicht zuletzt von dem zu lösenden Testproblem abhängig. Folglich erscheint die Bestimmung einer geeigneten Blockgröße ohne Kenntnis der Eingabe zur Installations- oder Compile-Zeit als nicht praktikabel. Die Auswahl einer geeigneten Blockgröße muss zur Laufzeit eines Programms stattfinden oder zumindest zu einem Zeitpunkt, zu dem die Eingabe des Programms bereits bekannt ist.

In diesem Kapitel wird ein Ansatz zur automatischen Auswahl geeigneter Blockgrößen für Implementierungsvarianten mit Loop-Tiling vorgestellt und dessen Integration in den Autotuning-Algorithmus beschrieben.

8.1. Das Problem der optimalen Blockgrößenauswahl

Die immer größer werdende Diskrepanz zwischen der Geschwindigkeit von Prozessoren und der Speicher verlangt nach einer effizienten Nutzung der Speicherhierarchie. Die Prozessoren mit hierarchischem Speichersystem verwenden Caches zum Verbergen der Latenzen von Hauptspeicherzugriffen. Nutzt ein Programm die Datenlokalität aus, so befinden sich viele Programmdaten während der Ausführung im Cache. Dadurch muss nur selten auf den langsamen Hauptspeicher zugegriffen werden.

Loop-Tiling ist eine wichtige Schleifentransformationstechnik zur Optimierung der Datenlokalität für Cachespeicher. Die Technik wurde 1969 von McKellar und Coffman eingeführt [MC69] und fand als erstes ihre Anwendung im Bereich des Compilerbaus [ASKL81, Wol89]. Die Idee von Loop-Tiling besteht darin, die Daten in Blöcke so zu zerlegen, dass diese jeweils in den Cache passen. Damit wird verhindert, dass die in den nachfolgenden Iterationen der Schleife zu nutzenden Daten aus dem Cache verdrängt werden. Die Technik des Loop-Tilings lohnt sich somit insbesondere für Schleifen, deren Arbeitsraum die Größe der Cache-Kapazität übersteigt.

Die Performance eines Loop-Tiling-Codes ist stark von der gewählten Blockgröße abhängig. Das **Problem der optimalen Blockgrößenauswahl (engl. Optimal Tile Size Selection (TSS))** ist das Problem der Ermittlung zulässiger Blockgrößen, die optimal in Bezug auf das jeweilige Kostenmodell sind [RU08]. Ein Kostenmodell besteht aus einer Kostenfunktion und einer oder mehreren Bedingungen zur Definition zulässiger Blockgrößen. Eine Kostenfunktion gibt an, welche der zulässigen Blockgrößen als optimal oder gut zu bewerten sind. Oft verwendete Kostenfunktionen sind die Anzahl der Cache-Fehlzugriffe oder die Laufzeit eines Programms. In [HK04] findet man eine Übersicht unterschiedlicher Algorithmen und Kostenmodelle zur Bestimmung von Blockgrößen bei Matrixmultiplikationsalgorithmen. Das Problem der optimalen Blockgrößenauswahl kann auch als ein Constraint-Satisfaction-Problem (CSP) formuliert werden [SS07].

Das TSS-Problem ist ein schwer zu lösendes Problem. Die Ursache dafür ist der komplexe Aufbau des Speichers moderner Rechnersysteme und die Abhängigkeit einer guten Blockgröße vom Speicherzugriffsverhalten einer Implementierung. Zudem hängt die Effizienz einer Blockgröße auch oft von den Eingabedaten ab. In der Literatur findet man eine Vielzahl von Ansätzen zur Lösung des TSS-Problems. Diese lassen sich grob in drei Gruppen aufteilen: analytische Ansätze, die auf einem Modell basieren [LRW91, BJWE92, CM99, CM95, YLR+05, SSF+12], empirische Ansätze [BACD97, WPD01, TH11] und modellbasierte empirische Ansätze [EGD+05, FCA05, Che07, FVP09].

Rein empirische Ansätze führen einen Loop-Tiling-Code mit einer großen Anzahl von Blockgrößen auf der Zielarchitektur aus und wählen dann die Blockgröße mit der schnellsten Laufzeit. Die Menge der auf der Zielarchitektur zu testenden Blockgrößen wird dabei oft durch eine Heuristik bestimmt. Die bekannte Autotuning-Bibliothek ATLAS [WPD01] führt zur Bestimmung optimaler Blockgrößen eine empirische Suche zur Installationszeit durch. Da die Problemgröße zur Installationszeit nicht bekannt ist, werden Blockgrößen für viele verschiedene Problemgrößen bestimmt. Generell ist es so, dass

empirische Ansätze zwar sehr gute Blockgrößen liefern, dafür aber viel Zeit brauchen, weil sie mit einem enorm großen Suchraum potenzieller Blockgrößen zu kämpfen haben.

Analytische Ansätze führen keine Laufzeitexperimente durch. Stattdessen werden geeignete Blockgrößen durch ein reines Kostenmodell bestimmt. Zur Generierung eines Modells werden oft die Informationen der statischen Codeanalyse (u. a. Erkennung von Wiederverwendungsmustern von Daten, Größe von Arbeitsräumen) und Kenntnisse über die relevanten Eigenschaften der Zielplattform (u. a. Cachegröße und Assoziativität, TLB-Größe, Netzwerklatenz) herangezogen. Der Entwurf analytischer Kostenmodelle gilt als schwierig, da er ein tiefgreifendes Verständnis der Funktionsweise von Rechnersystemen sowie der Interaktion zwischen Hardware- und Software-Komponenten erfordert.

In der Literatur gibt es zahlreiche Ansätze zur analytischen Blockgrößenbestimmung. Die Autoren in [BJWE92] versuchen, die Blockgröße so zu wählen, dass die Anzahl der Kapazitätsfehlzugriffe und der Schleifenoverhead minimiert wird. Die Blockgröße wird so gewählt, dass der Arbeitsraum, in der Publikation auch "Referenz-Fenster" genannt, in den Cache passt. Viel Arbeit wurde auch in die Berechnung der Größe von Arbeitsräumen investiert [Cla96, FST92a].

Andere bekannte analytische Ansätze berücksichtigen sowohl Kapazitätsfehlzugriffe als auch Konfliktfehlzugriffe [LRW91, CM95]. In [LRW91] wurde ein Modell zur Vorhersage von Selbstinterferenz-Konfliktfehlzugriffen entwickelt. Der Algorithmus in [LRW91] teilt ein zweidimensionales Array in Quadrate ein. Für jedes Quadrat wird dann die größtmögliche Blockgröße ermittelt, bei der die Quadratelemente, ohne Selbstinterferenz-Konflikte zu verursachen, in den Cache passen. Die von Coleman und McKinley in [CM95] vorgeschlagene Technik der Blockgrößenauswahl bestimmt die Blockgröße der rechteckigen Blöcke so, dass diese in den Cache passen und gleichzeitig die Anzahl von Fremdinterferenzen reduziert wird. Es wird zunächst eine Menge potentieller Blockgrößen mit Hilfe eines euklidischen Algorithmus ermittelt. Aus dieser Menge wird dann die Blockgröße ausgewählt, die eine gute Cacheausnutzung erlaubt und gleichzeitig die Anzahl von Fremdinterferenzen minimiert. In [YLR+05] wurde gezeigt, dass man für BLAS-Routinen mit Hilfe analytischer Modelle nahezu optimale Blockgrößen finden kann.

Die früheren Modelle der Blockgrößenauswahl haben nur den L1-Cache berücksichtigt [LRW91, BJWE92, CM95]. Die neueren Ansätze dagegen betrachten mehrere Ebenen der Cache-Hierarchie [SSF+12, MBY13]. Shirako et al. [SSF+12] kombinieren bei der Suche nach optimalen Blockgrößen zwei Modelle: das vorher existierende DL (Distinct Line)-Modell [FST92b] mit dem neu entworfenen optimistischen Modell ML (Minimum working set Lines). Die Kombination beider Modelle liefert eine untere und obere Schranke für die Wahl optimaler Blockgrößen. Dieser Ansatz kann dazu genutzt werden, den Suchraum potentieller Blockgrößen für die anschließende empirische Suche einzuschränken. Zur Bestimmung optimaler Blockgröße wird in [MBY13] die Wechselwirkung zwischen Loop-Tiling und Vektorisierung betrachtet.

Eine andere Herangehensweise zur Bestimmung geeigneter Blockgrößen bieten modellbasierte empirische Ansätze. Diese kombinieren eine empirische Suche mit einem analytischen Modell. Der Vorteil dabei ist, dass die Anzahl der zur Laufzeit evaluierten Blockgrößen klein bleibt. Gleichzeitig reichen die so gefunden Blockgrößen näher an das

Optimum heran als bei einem rein analytischen Vorgehen.

Die meisten Autotuner verwenden eine empirische [WPD01, BACD97, FJ05] oder eine modellbasierte empirische [TH11, CCH05, CSV11, FVP09] Suche. Das Autotuning-Framework in [CCH05] nutzt beispielsweise Compilermodelle und Suchheuristiken zur Bestimmung guter Blockgrößen.

Im nächsten Abschnitt wird untersucht, wie stark die Performance sequentieller Loop-Tiling-Varianten des betrachteten PC-Verfahrens vom RK-Type von der Wahl der Blockgröße abhängt.

8.2. Einfluss der Blockgröße auf die Laufzeit einer sequentiellen Implementierungsvariante

Die Konturdiagramme in Abb. 8.1 zeigen die Variation der Bereiche mit guten Blockgrößen auf zwei Systemen: AMD Opteron DP 246 und AMD Opteron 8350. Als Testproblem wurde das BRUSS2D-Problem verwendet. Die Konturdiagramme zeigen für unterschiedliche Systemgrößen n die relativen Laufzeiten der Implementierungsvariante *PipeDb2mt*, aufgetragen gegen die beste, über den Bereich der Blockgrößen $[1, 5000]$ erhaltene Laufzeit. Die Farbskalierung gibt die Qualität der Blockgrößen an. Die blauen Regionen entsprechen einer guten Wahl der Blockgröße, während die rote Regionen weniger gut geeignete Blockgrößen repräsentieren.

Für die Blockgrößen aus dem Bereich $[1, 5000]$ beträgt der Laufzeitunterschied auf beiden Rechnersystemen maximal 10 %. Auch auf anderen getesteten Rechnersystemen wurde ein ähnlicher Laufzeitunterschied bei der Verwendung der Blockgrößen aus dem Bereich $[1, 5000]$ beobachtet. Bei sehr großen Blockgrößen jedoch, die nahe an n heranreichen, kann der Performanceverlust bei mehr als 40 % liegen (s. Abb. 8.5 (b)).

Beim Betrachten der Laufzeitergebnisse auf dem AMD Opteron DP 246 System fällt auf, dass für manche Bereiche von Systemgrößen die Wahl einer Blockgröße aus dem Bereich $[1, 5000]$ keinen nennenswerten Einfluss auf die Laufzeit hat. Bei anderen Systemgrößen jedoch kann die Wahl einer ungünstigen Blockgröße aus dem Bereich $[1, 5000]$ zu einem Performanceverlust von bis zu 10 % führen. Auf dem AMD Opteron DP 246 System würde die Wahl kleiner Blockgrößen ≤ 700 die optimale Auswahlstrategie darstellen. Im Gegensatz dazu würde die Wahl kleiner Blockgrößen auf dem AMD Opteron 8350 Rechnersystem das Laufzeitverhalten um bis zu 9 % verschlechtern. Auf dem AMD Opteron 8350 wird die beste Laufzeit für die Implementierungsvariante *PipeDb2mt* mit der Wahl der Blockgröße nahe an 500 oder aus dem Bereich $[3000, 5000]$ erreicht.

Allgemein kann man festhalten, dass bei Varianten mit Loop-Tiling die Wahl einer geeigneten Blockgröße für das Erreichen einer hohen Performance notwendig ist.

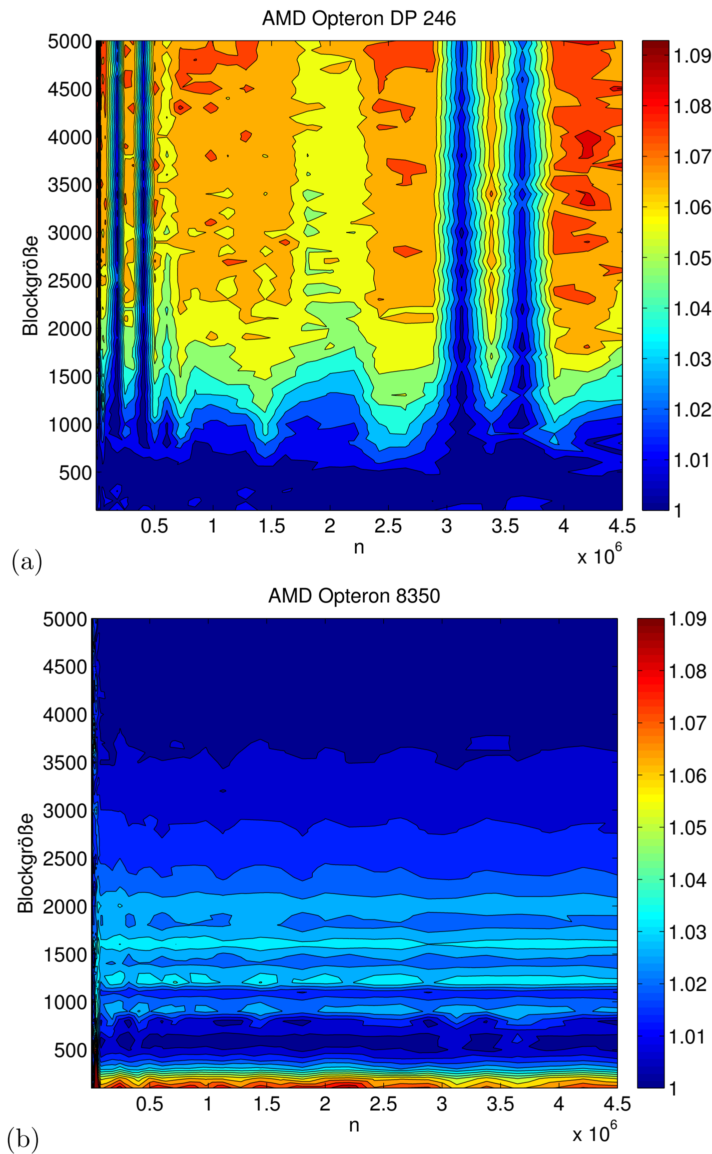


Abbildung 8.1.: Die normierte Laufzeit der *PipeDb2mt*-Implementierung in Abhängigkeit von der System- und Blockgröße für das BRUSS2D-Problem und das Radau IA (5)-Verfahren: (a) AMD Opteron 246, (b) AMD Opteron 8350.

8.3. Zielsetzung bei der automatischen Auswahl von Blockgrößen

Die Experimente im vorherigen Abschnitt zeigen, dass die Performance einer Implementierungsvariante mit einer Blockgröße nicht nur von den Eigenschaften der Zielplattform,

sondern auch von Problemparametern, wie z. B. der Systemgröße, abhängig ist. Da das Testproblem und seine Eigenschaften zur Compile- und Installationszeit nicht bekannt sind, muss die Auswahl von Blockgrößen für das Schleifen-Tiling zur Laufzeit stattfinden. Das Evaluieren vieler Blockgrößen während eines Programmlaufs ist kostspielig und erhöht die Gesamtprogrammlaufzeit. Die Anzahl zu testender Blockgrößen soll daher auf ein Minimum reduziert werden. Insbesondere gilt es zu vermeiden, dass Blockgrößen mit sehr langsamen Laufzeiten getestet werden.

Betrachtet man die Experimente in Abb. 8.1, so wird deutlich, dass der Suchraum viele Bereiche mit guten Blockgrößen enthält. Innerhalb dieser Bereiche beträgt der Laufzeitunterschied zwischen guten und weniger guten Blockgrößen nur einige wenige Prozent. Diese Beobachtung wurde auf allen getesteten Rechnern und für alle ODE-Probleme gemacht. Zum Erreichen einer hohen Performance wird es deshalb reichen, wenn man eine Blockgröße innerhalb dieser guten Bereiche bestimmen kann, anstatt viele Zeitschritte für die Evaluation von Blockgrößen zu spendieren, die die Laufzeit einer Implementierungsvariante nur um wenige Prozent verbessern würden. Man sollte auch bedenken, dass im Laufe des Autotuning-Prozesses die Blockgrößen möglichst für alle Loop-Tiling-Varianten bestimmt werden sollen. Führt man auf der Suche nach der optimalen Blockgröße mehrere Zeitschritte mit sehr langsamen Varianten durch, so leidet die Performance eines Autotuning-Algorithmus entsprechend.

Würde beispielsweise auf dem AMD Opteron DP 246 System für die Systemgrößen im Bereich $[1.0 \cdot 10^6, 2.5 \cdot 10^6]$ eine Blockgröße aus dem Bereich $[100, 1500]$ gewählt werden, so ist im Vergleich zur optimalen Blockgröße nur ein Performanceverlust von unter 5 % zu erwarten (Abb. 8.1 (a)). Für den gleichen Bereich der Systemgrößen auf dem AMD Opteron DP 8350 System würde die Wahl der Blockgrößen aus dem Bereich $[500, 5000]$ die Laufzeit nur um bis zu 4 % verschlechtern (Abb. 8.1 (b)). Der Unterschied zwischen den Blockgrößen in einem guten Bereich des Suchraumes ist relativ klein im Vergleich zum Laufzeitunterschied, der durch die Wahl einer langsamen Implementierungsvariante entstehen kann. Man betrachte dazu die Abbildung 8.2. Würde die langsamste Implementierungsvariante A gewählt werden, so würde der Performanceverlust 29 % betragen. Bei der Wahl der Implementierungen *D*, *PipeDe2m*, *PipeDb2m*, *ppDb1m*, *Dblock*, *PipeDe2m*, *ppDb1m* oder *PipeDb1m* würde der Laufzeitunterschied immer noch über 10 % betragen.

Aus diesem Grund hat die Methode der automatischen Auswahl von Blockgrößen das Ziel, eine gute Blockgröße in möglichst kurzer Zeit zu finden und nicht, die optimale Blockgröße zu bestimmen. Durch diese Zielsetzung hält man die Anzahl der zur Laufzeit evaluierter Blockgrößen klein und reduziert damit die Zeit zum Durchführen des Autotunings.

Der Code der Loop-Tiling-Varianten ist ein parametrisierter Code. Die Blockgrößen sind symbolische Konstanten. Diese können zur Laufzeit gesetzt werden. Somit kann der gleiche Code mit unterschiedlichen Blockgrößen ausgeführt werden. Die Notwendigkeit der Neugenerierung des Programmcodes entfällt und die Blockgrößen können zur Laufzeit evaluiert werden.

Als nächstes wird die Berechnungsstruktur des um eine Blockgrößenauswahl erweiter-

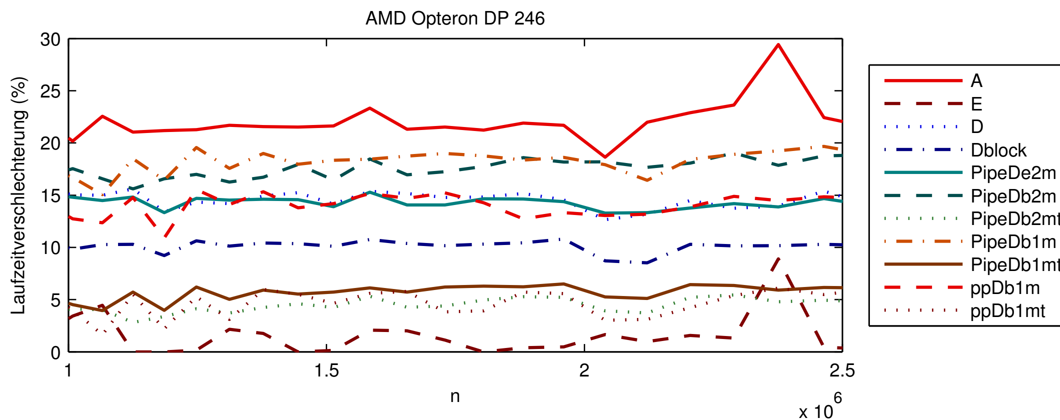


Abbildung 8.2.: Prozentualer Laufzeitunterschied zu der schnellsten Implementierungsvariante E für n aus dem Bereich $[1.0 \cdot 10^6, 2.5 \cdot 10^6]$, das BRUSS2D-Problem und das Radau IA (5)-Verfahren.

ten dynamischen Autotuning-Algorithmus präsentiert.

8.4. Der selbstadaptive ODE-Solver mit integrierter Blockgrößenauswahl

In diesem Abschnitt wird die Funktionsweise des um die Blockgrößenauswahl erweiterten selbstadaptiven ODE-Solvers beschrieben. Dieser bestimmt zur Laufzeit die schnellste Implementierungsvariante aus dem Kandidatenpool für das zu lösende AWP und die gegebene Zielarchitektur. Außerdem werden geeignete Blockgrößen für die Implementierungsvarianten mit Loop-Tiling gewählt.

8.4.1. Die Zeitschritt-Phasen des selbstadaptiven ODE-Solvers

Der selbstadaptive ODE-Solver nutzt die zeitschrittorientierte Berechnungsstruktur der ODE-Lösungsverfahren aus, um zur Laufzeit die schnellste Implementierungsvariante aus dem Kandidatenpool auszuwählen.

Die zeitschrittorientierte Berechnungsstruktur des adaptiven ODE-Solvers besteht aus vier Phasen:

1. **Vorauswahlphase.** Die Vorauswahlphase findet nach dem Solveraufruf statt, aber noch vor dem ersten Zeitschritt des Integrationsprozesses. Zu diesem Zeitpunkt ist das zu lösende AWP bereits bekannt. In der Vorauswahlphase werden die Implementierungsvarianten aus dem Kandidatenpool entfernt, die auf das zu lösende ODE-Problem nicht anwendbar sind. Derzeit können die spezialisierten Implementierungsvarianten nur auf ODE-Probleme mit einer beschränkten Zugriffsdistanz angewandt werden. Deshalb wird in der Vorauswahlphase überprüft, ob das zu

lösende ODE-System eine beschränkte Zugriffsdistanz besitzt. Wenn ja, dann werden die spezialisierten Implementierungsvarianten im Pool beibehalten. Andernfalls werden die spezialisierten Implementierungsvarianten aus dem Kandidatenpool entfernt.

2. **Modellbasierte Blockgrößenvorauswahlphase.** In dieser Phase, die ebenfalls vor dem ersten Zeitschritt erfolgt, wird für jede Implementierungsvariante aus dem Kandidatenpool eine kleine Menge potentiell effizienter Blockgrößen bestimmt. Dies geschieht anhand eines analytischen Modells. Das Modell basiert auf der Berechnung von Arbeitsräumen von Schleifen und berücksichtigt bei der Wahl der Blockgröße die Eigenschaften der Cache-Hierarchie der Zielplattform.

3. **Autotuningphase.** Die Autotuningphase findet während der ersten Zeitschritte des Integrationsprozesses statt. Die Anzahl der Autotuning-Schritte hängt von der Anzahl zu evaluierender Implementierungsvarianten und Blockgrößen ab. In dieser Phase werden nacheinander alle Implementierungsvarianten aus dem Kandidatenpool mit den für sie in der letzten Phase bestimmten Blockgrößen auf der Zielarchitektur ausgeführt. Für jede Implementierungsvariante und jede evaluierte Blockgröße wird gesondert die Laufzeit eines Zeitschrittes aufgezeichnet. Am Ende der Autotuningphase wird die schnellste Implementierungsvariante und die beste Blockgröße anhand der erfassten Laufzeiten ermittelt.

4. **Berechnungsphase.** Die restlichen Zeitschritte werden mit der Implementierungsvariante und der Blockgröße ausgeführt, die in der Autotuningphase zum besten Ergebnis mit der schnellsten Laufzeit geführt haben.

Der Pseudocode des selbstadaptiven Zeitschrittalgorithmus ist in Abbildung 8.3 gegeben.

Als nächstes wird die modellbasierte Blockgrößenvorauswahlphase beschrieben.

8.4.2. Modellbasierte Blockgrößenvorauswahlphase

Wie bereits im Abschnitt 8.3 diskutiert wurde, müssen die Blockgrößen aufgrund der Abhängigkeit von Eingabe, zur Laufzeit ausgewertet werden. Bei dem im Abschnitt 7.5 vorgestellten Autotuning-Algorithmus wird die beste Implementierungsvariante während der ersten Zeitschritte des Integrationsprozesses bestimmt. Deshalb muss auch die Auswahl von Blockgrößen während der ersten Zeitschritte erfolgen, um in den bestehenden Online-Autotuning-Algorithmus erfolgreich integriert werden zu können.

Bei der Blockgrößenauswahl ist das Ziel gesetzt worden, eine gute Blockgröße in möglichst kurzer Zeit automatisch auszuwählen (vgl. Abschnitt 8.3). Ein Problem dabei ist jedoch, dass der Suchraum potenzieller Blockgrößen außerordentlich groß sein kann. Eine erschöpfende Suche ist in der Regel nicht praktikabel. Die meisten Autotuner verwenden deshalb approximative Methoden, um nach einer Blockgröße zu suchen, die einer optimalen Blockgröße möglichst nahe kommt. Aber sogar eine approximative Bestimmung einer Blockgröße würde eine große Anzahl von Zeitschritten erfordern, wenn der Suchraum nicht zuvor erheblich eingegrenzt werden würde. Da im Kandidatenpool mehrere

```
1: Sei P die zu lösende Probleminstanz;
2: Sei C die Kandidatenmenge der Implementierungsvarianten;

   // 1. Vorauswahlphase

3: C_1 ← pre-select (C, P);                       // Vorauswahl von Implementierungsvarianten

   // 2. Modellbasierte Blockgrößenvorauswahlphase

4: B ← select_tile_size_samples(C_1);

   // 3. Autotuningphase

5: t ← t_0;                                       // Setze t auf Anfang des Integrationsintervalls
6: h ← h_init;                                    // Setze Anfangsschrittweite
7: T_best ← ∞;                                    // Laufzeit der bis jetzt besten Implementierung
8: Wähle eine (beliebige) Implementierung ohne Loop-Tiling c ∈ C_1;
9: ts ← n;                           // Blockgröße wird nicht benötigt, nutze n als Dummy
10: compute_time_step(c, ts, t);                  // Aufwärmschritt
11: Führe Schrittweitenkontrolle durch;
12: Aktualisiere t, wenn der Fehler klein genug;

13: while (C_1 ≠ ∅)
14: {
15:     Wähle eine (beliebige) Implementierung c ∈ C_1;
16:     C_1 ← C_1 \ c;
17:     TS ← set_of_tile_sizes(B, c);

18:     while (TS ≠ ∅ ∧ t < t_e)
19:     {
20:         Wähle ts ∈ TS;
21:         TS ← TS \ ts;
22:         T ← compute_time_step(c, ts, t);
23:         if (T < T_best)
24:         (c_best, T_best, ts_best) ← (c, T, ts);  // Implementierung c mit der Blockgröße ts ist
25:         // die schnellste bis jetzt
26:         Führe Schrittweitenkontrolle durch;
27:         Aktualisiere t, wenn der Fehler klein genug;
28:     }
29: }

   // 4. Führe die restlichen Schritte mit der schnellsten Implementierungsvariante aus

30: while (t < t_e)                              // Verbleibende Teil des Integrationsintervalls
31: {
32:     compute_time_step(c_best, ts_best, t);
33:     Führe Schrittweitenkontrolle durch;
34:     Aktualisiere t, wenn der Fehler klein genug;
35: }
```

Abbildung 8.3.: Der selbstadaptive Zeitschrittalgorithmus mit integrierter Blockgrößenauswahl.

Implementierungsvarianten mit Loop-Tiling enthalten sind, für welche effiziente Blockgrößen ermittelt werden sollten, sind Methoden zur Reduzierung der Anzahl der zur Laufzeit zu evaluierenden Blockgrößen notwendig.

Generell kann die Anzahl in der Autotuningphase auszuwertender Blockgrößen und folglich auch die Anzahl für die Blockgrößenauswahl benötigter Zeitschritte entweder durch den Einsatz von Heuristiken oder durch das Verwenden eines analytischen Modells reduziert werden. In dieser Arbeit wird ein analytisches Modell verwendet, um die Anzahl der zur Laufzeit auszuwertenden Blockgrößen zu reduzieren.

Anforderungen an das Blockgrößenauswahlmodell

An das Modell sind folgende Anforderungen gestellt worden:

- Es soll präzise genug sein, um eine Auswahl guter Blockgrößen zu gewährleisten.
- Das Modell soll nur einen geringen Zeitoverhead erfordern.
- Das Modell soll so konstruiert sein, dass man es, möglichst ohne großen Aufwand, für andere ODE-Verfahren erweitern und nutzen kann.

Das entstandene Modell wird im folgenden Abschnitt beschrieben.

Das Blockgrößenvorauswahlmodell

Das Modell basiert auf der Bestimmung von Arbeitsräumen der in den Implementierungsvarianten auftretenden Schleifen und berücksichtigt bei der Wahl einer Blockgröße die relevanten Eigenschaften der Speicherhierarchie der Zielarchitektur. Das Modell bestimmt eine kleine Menge potentiell effizienter Blockgrößen, die dann später in der Autotuningphase auf der Zielarchitektur evaluiert werden.

Die Idee der Loop-Tiling-Technik besteht darin, die Wiederverwendung von Daten zu maximieren und die Zugriffe auf den relativ langsamen Hauptspeicher zu reduzieren. Die Blockgröße sollte deshalb so gewählt werden, dass der Arbeitsraum eines Blocks, d. h. die Menge aller innerhalb des Blocks referenzierten Datenelemente, in den Cache passt. Wegen der Komplexität moderner Speicherhierarchien ist es jedoch nicht offensichtlich, für welche Cache-Stufe die Blockgröße optimiert werden soll. Eine nahezu optimale Performance kann aber häufig durch die Optimierung der Blockgröße auf die schnellste Cache-Stufe der Speicherhierarchie erreicht werden.

Zur Erinnerung: Ein Arbeitsraum eines Programmabschnittes ist definiert als die Menge aller innerhalb dieses Programmabschnittes referenzierten Datenelemente. Die Arbeitsräume eines Programms entsprechen üblicherweise den Iterationen von Schleifen. Die Implementierungsvarianten expliziter PC-Verfahren vom RK-Type enthalten mehrere Schleifen. Jede dieser Schleifen stellt einen Arbeitsraum dar. Bei der Entwicklung eines analytischen Modells stellt sich daher die Frage, welche Arbeitsräume einen großen Einfluss auf die Laufzeit einer Implementierungsvariante ausüben und deshalb für die Blockgrößenauswahl von großer Bedeutung sind. Diese Frage ist nicht einfach zu beantworten. Oft sind aber die Arbeitsräume laufzeitrelevant, auf die in einem Programm

häufig zugegriffen wird. Damit eine Implementierungsvariante effizient arbeiten kann, sollen sie nach Möglichkeit im Cache zwischengespeichert werden.

Generell werden bei der Blockgrößenauswahl natürlich nur die Arbeitsräume betrachtet, deren Größe von der Blockgröße B abhängig ist. Das sind Schleifen, deren Iterationsraum entweder selbst in Blöcke der Blockgröße B aufgeteilt worden ist oder deren innere Schleifen über die Blöcke der Blockgröße B iterieren. Die Blockgröße wird nur für die Implementierungsvarianten aus dem Kandidatenpool gewählt, die von der Blockgröße B abhängige Arbeitsräume enthalten. Das sind die Implementierungsvarianten *PipeDb2m*, *PipeDb2mt*, *Dblock*, *PipeDb1m*, *PipeDb1mt*, *ppDb1m* und *ppDb1mt*.

Der wichtigste und der größte Arbeitsraum einer Implementierungsvariante ist der Arbeitsraum eines Zeitschrittes WS_{ts}. Wird die Schrittweitenkontrolle als ein Teil des Zeitschrittes aufgefasst, so enthält dieser Arbeitsraum alle im Programm referenzierten Datenelemente und ermöglicht deren Wiederverwendung innerhalb aufeinanderfolgender Zeitschritte. Kann der gesamte Arbeitsraum eines Zeitschrittes einer Implementierungsvariante im Cache gespeichert werden, so sollten keine Kapazitätsfehlzugriffe auftreten. Ist die Cachegröße jedoch kleiner als die Größe des Arbeitsraumes eines Zeitschrittes, so sind andere Arbeitsräume ausschlaggebend für die Laufzeit, damit zumindest eine Teilmenge der Daten eines Zeitschrittes wiederverwendet werden kann. Der nächstgrößere Arbeitsraum der Implementierungsvarianten *PipeDb2mt*, *PipeDb2m*, *Dblock*, *PipeDb1mt* und *PipeDb1m* ist der Arbeitsraum eines Korrektorschrittes WS_{cs}, der die Wiederverwendung der Daten innerhalb aufeinanderfolgender Korrektorschritte erlaubt. Bei den Implementierungsvarianten *ppDb1m* und *ppDb1mt* ist der nächstgrößere Arbeitsraum der Arbeitsraum eines Pipelining-Schrittes. Insbesondere ist bei Pipelining-Varianten der Arbeitsraum eines Pipelining-Schrittes in der langen Sweep-Phase WS_{ps} wichtig. Dieser Arbeitsraum entspricht einem diagonalem Lauf über die Korrektorschritte in der Sweep-Phase und ermöglicht die Wiederverwendung von Daten zwischen aufeinanderfolgenden Iterationen der Systemdimensionsschleife j. Die Arbeitsräume eines Pipelining-Schrittes der Initialisierungs- und Finalisierungsphase können dagegen bei der Blockgrößenauswahl ignoriert werden, da diese nur wenige Iterationen der über die Systemdimension laufenden Schleife j erfordern. Ist die Kapazität eines Caches kleiner als die Größe des Arbeitsraumes eines Pipelining- oder Korrektorschrittes, so kann die Wiederverwendung von Daten nur innerhalb aufeinanderfolgender Iterationen der $i-$ oder der l-Schleife oder auch innerhalb der durch Tiling entstandenen Schleifen stattfinden. Bei noch kleineren Caches kann nur die Wiederverwendung von kleinen Teilen eines Arbeitsraumes in Betracht kommen.

Für die Implementierung eines analytischen Modells wurden auf Grund ihres großen Einflusses auf die Laufzeit der Arbeitsraum eines Zeitschrittes WS_{ts} und der Arbeitsraum eines Korrektorschrittes bzw. – für die Pipelining-Varianten – der Arbeitsraum eines Pipelining-Schrittes gewählt. Zusätzlich zu diesen Arbeitsräumen wurden für jede Variante noch maximal vier kleinere laufzeitrelevante Arbeitsräume (WS_1, WS_2, WS_3, WS_4) gewählt. Diese entsprechen im Wesentlichen den Arbeitsräumen der i-, j- und l-Schleifen sowie dem Arbeitsraum der Größe $2B$, der durch das Loop-Tiling der über die Systemdimension laufenden Schleife entstanden ist.

Die Größe eines Arbeitsraumes ergibt sich als Funktion der Systemgröße n, der Stufenanzahl s, der Anzahl der Korrektorschritte m, der Zugriffsdistanz $d(\mathbf{f})$ und der Blockgröße ts. Die einzelnen Arbeitsräume lassen sich aus der Programmstruktur herleiten. Zur Bestimmung des Arbeitsraumes eines Schleifenkomplexes bestimmt man zuerst die Menge der Datenelemente, die innerhalb der innersten Schleifen des Komplexes referenziert werden. Danach bildet man die Vereinigung dieser Menge mit der Menge der Datenelemente, die in umschließenden Schleifen referenziert werden. Dieser Vorgang wird so lange wiederholt, bis die oberste Verschachtelungsebene erreicht ist.

Die Tabelle 8.1 zeigt eine Übersicht ausgewählter Arbeitsräume und deren Größe für alle Implementierungsvarianten im Kandidatenpool, die Loop-Tiling verwenden. In der Tabelle bezeichnet ts die Blockgröße, n ist die Größe des ODE-Systems, s ist die Anzahl der Stufen des ODE-Verfahrens, m ist die Anzahl der Korrektorschritte und $d(\mathbf{f})$ ist die Zugriffsdistanz des ODE-Systems.

Das Blockgrößenauswahlmodell geht von einem allgemeinen Cache-Modell aus. Es wird angenommen, dass der Cache inklusiv aufgebaut ist, d. h. eine Cachezeile, die im L1-Cache ist, ist auch im L2- und L3-Cache vorhanden. Weiter wird angenommen, dass die Caches voll-assoziativ sind und LRU als Ersetzungsstrategie nutzen, d.h. alle entstehenden Cache-Fehlzugriffe sind entweder auf einen Capacity- oder ein Cold-Miss zurückzuführen.

Die Wahl einer Blockgröße erfolgt abhängig von der Größe des ODE-Systems n, der Kapazität der Caches $i = 1, \dots, r$ der Zielarchitektur, die als $C(\mathrm{L1}), C(\mathrm{L2}), \dots, C(\mathrm{L}r)$ gegeben ist, sowie der Blocklänge des L1-Caches $C_b(L_1)$. Die Cachekapazität und die Blockgröße sind durch die Anzahl der Gleitkommawerte gegeben.

Der Einfachheit halber wurde bei der Bestimmung von Arbeitsräumen in Tab. 8.1 nur der Speicherplatzbedarf für Vektorelemente berücksichtigt, nicht aber der Speicherplatzbedarf für weitere Daten, wie z. B. die Verfahrenskoeffizienten, skalare Variablen oder Daten auf dem Stack, die oft im Programm gebraucht werden, dafür aber wenig Speicher benötigen. Um die daraus eventuell entstehende Ungenauigkeiten im Modell zu vermeiden, wurde für die Cachekapazität eine geringere Größe angenommen. Es werden 90 % der physikalischen Kapazität eines Caches genutzt. Die effektive Größe eines Caches der Stufe $i = 1, \dots, r$ wird als $C_{\mathrm{eff}}(\mathrm{L}i) = C(\mathrm{L}i) \cdot 0.9$ definiert. Auf dieser Weise wird sichergestellt, dass die Größe des Arbeitsraumes, für den eine Blockgröße gewählt wird, die Kapazität der jeweiligen Cache-Stufe nicht übersteigt.

Um den Suchraum potenzieller Blockgrößen einzuschränken, die in der Autotuningphase evaluiert werden, wird zunächst für jede Implementierungsvariante mit Hilfe des Modells eine kleine Menge von Blockgrößen bestimmt. Die Strategie zur Auswahl geeigneter Blockgrößen lautet wie folgt:

> Bestimme die Blockgröße so, dass jeder Arbeitsraum, der einen großen Einfluss auf die Laufzeit ausübt, in der schnellstmöglichen Cache-Stufe gespeichert werden kann.

Diese Strategie ähnelt der Idee in [BJWE92] in der Art und Weise, dass man einerseits

ID	Schleife	Größe
Dblock		
WS_{ts}	Zeitschritt	$(2s+3)n+ts$
WS_{cs}	Korrektorschritt	$(2s+1)n+ts$
WS_1	Eine Iter. der i Schleife	$(s+2)n+2d(\mathbf{f})+ts$
WS_2	Eine Iter. der j-Schleife $(k=1,\ldots,m-2)$	$(s+3)ts+2d(\mathbf{f})$
WS_3	Eine Iter. der l-Schleife $[(jj)]$	$2\cdot ts$
PipeDb2mt		
WS_{ts}	Zeitschritt	$(2s+3)n+ts$
WS_{cs}	Korrektorschritt	$(2s+1)n+ts$
WS_1	Eine Iter. der j Schleife $(k=1,\ldots,m-2)$	$(2s+3)ts+s\cdot 2d(\mathbf{f})$
WS_2	Eine Iter. der i-Schleife	$(s+3)ts+2d(\mathbf{f})$
WS_3	Eine Iter. der l-Schleife $[(jj)]$	$2\cdot ts$
PipeDb2m		
WS_1	Eine Iter. der j Schleife $(k=1,\ldots,m-2)$	$(2s+2)ts+s\cdot 2d(\mathbf{f})$
WS_2	Eine Iter. der i-Schleife	$(s+2)ts+2d(\mathbf{f})$
PipeDb1m		
WS_{ts}	Zeitschritt	$(s(2m-2))ts+(s+3)n$
WS_{cs}	Korrektorschritt	$(s+1)\cdot n+s\cdot 2ts$
WS_1	Eine Iter. der j-Schleife $(k=1,\ldots,m-2)$	$(2s+2)ts+s\cdot 2d(\mathbf{f})$
WS_2	Eine Iter. der i-Schleife	$(s+2)ts+2d(\mathbf{f})$
PipeDb1mt		
WS_{ts}	Zeitschritt	$(s(2m-2)+1)ts+(s+3)n$
WS_{cs}	Korrektorschritt	$(s+1)\cdot n+s\cdot 2ts+ts$
WS_1	Eine Iter. der j-Schleife $(k=1,\ldots,m-2)$	$(2s+3)ts+s\cdot 2d(\mathbf{f})$
WS_2	Eine Iter. der i-Schleife	$(s+3)ts+2d(\mathbf{f})$
WS_3	Eine Iter. der l-Schleife $[(jj)]$	$2\cdot ts$
ppDb1m		
WS_{ts}	Zeitschritt	$(s(2m-2))ts+(s+3)n$
WS_{ps}	Pipelining-Schritt	$((3s+1)m+3)ts$
WS_1	Eine Iter. der j-Schleife $(k=1,\ldots,m-2)$	$(2s+2)ts+s\cdot 2d(\mathbf{f})$
WS_2	Eine Iter. der j-Schleife $(k=m-1)$	$(2s+1)ts+s\cdot 2d(\mathbf{f})$
WS_3	Eine Iter. der i-Schleife	$(s+2)ts+2d(\mathbf{f})$
ppDb1mt		
WS_{ts}	Zeitschritt	$(s(2m-2)+1)ts+(s+3)n$
WS_{ps}	Pipelining-Schritt	$((3s+1)m+4)ts$
WS_1	Eine Iter. der j-Schleife $(k=1,\ldots,m-2)$	$(2s+3)ts+s\cdot 2d(\mathbf{f})$
WS_2	Eine Iter. der j-Schleife $(k=m-1)$	$(2s+2)ts+s\cdot 2d(\mathbf{f})$
WS_3	Eine Iter. der i-Schleife	$(s+3)ts+2d(\mathbf{f})$
WS_4	Eine Iter. der l-Schleife $[(jj)]$	$2\cdot ts$

Tabelle 8.1.: Ausgewählte Arbeitsräume für Implementierungsvarianten mit Loop-Tiling.

versucht, die Arbeitsräume eines Programms im Cache zu halten, um die Anzahl der Kapazitätsfehlzugriffe zu reduzieren, und andererseits für die in die Cache-Hierarchie passenden Arbeitsräume die Blockgröße so groß wie möglich wählt, damit der Schleifen-Overhead minimiert wird.

Die Vorschrift der modellbasierten Vorauswahl von Blockgrößen lautet wie folgt:

1. Identifiziere für jede zu betrachtende Implementierungsvariante die Menge laufzeitrelevanter Arbeitsräume $\mathcal{W}$.

2. Für jeden Arbeitsraum $w \in \mathcal{W}$ und jede Cache-Stufe i berechne $LT(w, C_{\text{eff}}(\text{L}i))$. Dies ist die größtmögliche Blockgröße, für die der Arbeitsraum w in den Cache der Stufe i passt. Passt der Arbeitsraum w nicht in die Cache-Stufe i, so wird $LT(w, C_{\text{eff}}(\text{L}i))$ auf die Größe des ODE-Systems n gesetzt.

$$LT(w, C_{\text{eff}}(\text{L}i)) = \begin{cases} \max\{ts \in \{1, \dots, \min(n, C(\text{L}i))\} \mid \\ \qquad \text{ws_size}(w, ts) \le C_{\text{eff}}(\text{L}i)\}, \\ \qquad \text{wenn } \exists\, ts : \text{ws_size}(w, ts) \le C_{\text{eff}}(\text{L}i), \\ n, \quad \text{sonst.} \end{cases} \tag{8.1}$$

Die Hilfsfunktion $\text{ws_size}(I, ts)$ berechnet die Größe des Arbeitsraumes w mit der Blockgröße ts und ist wie folgt definiert:

$$\text{ws_size}(w, ts) = \begin{cases} \text{Größe des Arbeitsraumes } w & \\ \text{mit der Blockgröße } ts, & \text{wenn } ts \in \{1, \dots, n\}, \\ n, & \text{sonst.} \end{cases} \tag{8.2}$$

3. Bilde das Minimum aller Blockgrößen, die im vorherigen Schritt berechnet wurden, ts_{min}. Nehme ts_{min} in die Menge der Blockgrößenvorschläge $\mathcal{TS}$ auf. Durch die Wahl von ts_{min} wird garantiert, dass jeder Arbeitsraum in der schnellstmöglichen Cache-Stufe gespeichert werden kann, in die er hineinpasst.

4. Sei n_{line} die vorgegebene Anzahl von L1-Cachezeilen. Ist ts_{min} deutlich größer als $\#lines \cdot C_b(L_1)$, so nehme auch $ts_{small} = \#lines \cdot C_b(L_1)$ in die Menge der Blockgrößenvorschläge $\mathcal{TS}$ auf.

Durch die Minimumstrategie wird der größte Arbeitsraum in die schnellstmögliche Cache-Stufe platziert, in die er noch hineinpasst. Gleichzeitig wird aber die gleiche Blockgröße auch für andere Arbeitsräume verwendet, für die man u. U. eine größere Blockgröße wählen könnte.

Den Vorschlag für die zweite Blockgröße (siehe Abb. 8.4, Zeile 6) liefert eine Heuristik, die auf praktischer Erfahrung basiert. Auf manchen Rechnern, wie z. B. auf dem AMD Opteron DP 246, wurde beobachtet, dass kleine Blockgrößen, die einer kleinen Cachezeilenanzahl entsprechen (z. B. $\#lines = 16$), oft eine bessere Performance erreichen als Blockgrößen, die einen größeren Bereich eines Caches nutzen.

1: Sei $\mathcal{W} = \{WS_{TS}, WS_{CS}, WS_1, WS_2, \ldots WS_j\}$ die Menge laufzeitrelevanter Arbeitsräume.
2: Sei $\mathcal{C} = \{\mathrm{L1}, \mathrm{L2}, \ldots \mathrm{L}i\}$ die Menge der Cache-Stufen $i = 1, \ldots, r$.
3: Sei $\mathcal{TS}$ die Menge zu bestimmender Blockgrößen.
4: $ts_{min} \leftarrow \min\{LT(w, C_{\text{eff}}(L_i)) \,|\, w \in \mathcal{W}, L_i \in \mathcal{C}\}$;
5: $\mathcal{TS} \leftarrow \{ts_{min}\}$;
6: **if** $(ts_{min} \geq \#lines \cdot C_b(L_1) + 100)$ $ts_{small} \leftarrow \#lines \cdot C_b(L_1)$;
7: $\mathcal{TS} \leftarrow \mathcal{TS} \cup \{ts_{small}\}$;

Abbildung 8.4.: Modellbasierte Vorauswahl von Blockgrößen für allgemeine Implementierungsvarianten mit Loop-Tiling.

Bisher wurde die Methode der Blockgrößenauswahl für allgemeine Implementierungsvarianten mit Loop-Tiling beschrieben. Diese ist in Abbildung 8.4 gegeben.

Zur Auswahl von Blockgrößen für spezialisierte Varianten wird ein ähnlicher Ansatz wie der in Abbildung 8.4 verfolgt. Aufgrund der besonderen Struktur spezialisierter Varianten müssen die gewählten Blockgrößen zusätzlichen Bedingungen genügen:

- Die gewählte Blockgröße muss größer oder gleich der Zugriffsdistanz des ODE-Systems sein, d. h. sie muss der Bedingung $ts \geq d(\mathbf{f})$ genügen. Damit wird sichergestellt, dass zur Berechnung eines Blocks des Korrektorschrittes k maximal drei Blöcke des Korrektorschrittes $k-1$ notwendig sind.

- Bei Pipelining-Varianten bezeichnet ts diejenige Blockgröße, die das ODE-System der Größe n in $\lfloor n/ts \rfloor$ Blöcke aufteilt. Weil für den Aufbau einer vollen Pipeline mindestens m Blöcke notwendig sind, muss auch das ODE-System der Größe n in mindestens m Blöcke aufgeteilt werden. Für Pipelining-Varianten vorgeschlagene Blockgrößen müssen daher eine weitere Bedingung erfüllen: $ts \leq \lfloor n/m \rfloor$.

Für spezialisierte Implementierungsvarianten ist die Heuristik in Zeile 6 in Abb. 8.4 durch die Bedingung $d(\mathbf{f}) \geq \#lines \cdot C_b(L_1) + 100$ ersetzt worden. Ist diese erfüllt, so wird $d(\mathbf{f})$ und ansonsten $\#lines \cdot C_b(L_1)$ in die Menge empirisch zu evaluierender Blockgrößen $\mathcal{TS}$ aufgenommen. Die Zugriffsdistanz $d(\mathbf{f})$ wird als zweite Blockgröße vorgeschlagen, weil es die kleinstmögliche Blockgröße ist, die für spezialisierte Implementierungsvarianten gewählt werden kann. Die Zugriffsdistanz wird aber nur dann in die Menge empirisch zu evaluierender Blockgrößen aufgenommen, wenn sie nicht zu klein und hinreichend größer als ts_{min} ist.

Am Ende liefert der modellbasierte Ansatz eine Menge von Blockgrößen $\mathcal{TS}$, die für jede Loop-Tiling-Variante maximal zwei Blockgrößen enthält. Die Blockgrößen aus dieser Menge werden in der Autotuningphase auf der Zielarchitektur evaluiert und für jede Variante wird die beste Blockgröße anhand der gemessenen Laufzeiten ermittelt.

8.4.3. Realisierung der Autotuningphase

Die im Kandidatenpool enthaltenen Implementierungsvarianten unterscheiden sich zwar in der Schleifenreihenfolge und den Datenstrukturen, besitzen aber die gleichen numerischen Eigenschaften. Dadurch können unterschiedliche Implementierungsvarianten für die Berechnung einzelner Zeitschritte genutzt werden.

In der Autotuningphase werden nacheinander alle Implementierungsvarianten aus dem Kandidatenpool zur Berechnung von einem oder mehreren Zeitschritten herangezogen. Die Implementierungsvarianten ohne Loop-Tiling werden für die Berechnung je eines Zeitschrittes verwendet. Eine Loop-Tiling-Variante führt je einen Zeitschritt mit einer Blockgröße aus, bis alle für diese Variante vom Modell bestimmten Blockgrößen getestet sind. Daher entspricht die Gesamtanzahl der mit einer Loop-Tiling-Variante durchgeführten Zeitschritte der Anzahl von Blockgrößen in der modellbestimmten Menge. Die Laufzeit einer Implementierungsvariante mit der jeweiligen Blockgröße wird mit Hilfe eines Protokollierungsmechanismus erfasst. Anhand des Vergleichs aufgezeichneter Laufzeiten wählt der Autotuner zunächst für jede Implementierungsvariante die beste Blockgröße aus und dann die beste Implementierungsvariante. Die Einzelheiten der Laufzeiterfassung sind dem Abschnitt 7.6 zu entnehmen.

Im Allgemeinen hängt die Performance einer Implementierung vom Anfangszustand des Rechnersystems ab, insbesondere vom Inhalt der Cache-Hierarchie. Da beim Programmstart die Caches erstmals mit Programmdaten gefüllt werden müssen, ist die Zeit für die Berechnung eines Zeitschrittes zu Beginn der Ausführung besonders hoch. Sobald aber die Daten in die Cache-Hierarchie geladen sind, sinkt diese. Um also eine repräsentative Laufzeit für eine Implementierungsvariante zu erhalten, muss die Laufzeit eines Zeitschrittes ausgehend von einem aufgewärmten Zustand der Cache-Hierarchie gemessen werden. Damit die Cache-Hierarchie so reproduzierbar wie möglich aufgewärmt wird, wurde beim Grundalgorithmus im Kapitel 7.5 zunächst zwischen akzeptierten und verworfenen Zeitschritten unterschieden (vgl. Abb. 8.3). Die Laufzeit einer Implementierungsvariante wurde erst für einen akzeptierten Zeitschritt gemessen, der unmittelbar auf einen vorausgehenden akzeptierten Zeitschritt folgt. Als Ergebnis dieser Strategie mussten mehrere Aufwärmschritte ausgeführt werden, dadurch verlängerte sich die Dauer der Autotuningphase.

Um zu sehen, ob man die Dauer der Autotuningphase eventuell verkürzen kann, wurde eine experimentelle Analyse durchgeführt, in der der Einfluss des Cacheaufwärmens auf die Laufzeit von Zeitschritten bei unterschiedlichen Implementierungsvarianten untersucht worden ist. Außerdem wurde untersucht, wie stark ein verworfener Zeitschritt die Laufzeit eines unmittelbar danach folgenden akzeptierten Zeitschrittes beeinflusst.

Die experimentelle Untersuchung hat gezeigt, dass vor allem der erste Zeitschritt für das Aufwärmen der Cache-Hierarchie maßgeblich ist. Der Laufzeitunterschied des ersten Zeitschrittes zu einem nachfolgenden Zeitschritt beträgt mehr als $\gtrsim 2\,\%$. Für die restlichen Zeitschritte der Autotuningphase konnte nur ein geringer Laufzeitunterschied zwischen einem aufgewärmten und nicht aufgewärmten Zeitschritt festgestellt werden. Die Experimente haben auch gezeigt, dass der Einfluss eines verworfenen Zeitschrittes auf die Laufzeit eines unmittelbar danach folgenden akzeptierten Zeitschrittes eher ge-

ring ist. Deshalb wurde der Autotuning-Algorithmus so modifiziert, dass nur der erste Integrationszeitschritt zum Aufwärmen von Cache-Hierarchie benutzt wird. Eine weitere Änderung besteht darin, dass man bei der Bestimmung der Laufzeit einer Implementierungsvariante nicht mehr zwischen akzeptierten und verworfenen Zeitschritten unterscheidet. Die Bestimmung der Laufzeit einer Implementierungsvariante geschieht unabhängig davon, ob der Zeitschritt verworfen oder akzeptiert wurde. Bei einem verworfenen Zeitschritt wird die Zeit zur Wiederherstellung des alten Approximationsvektors $\boldsymbol{\eta}$ nicht mitgemessen.

Allgemein hängt die Anzahl der Autotuning-Schritte von der Anzahl evaluierter Implementierungsvarianten und Blockgrößen ab. Für ODE-Systeme mit einer beschränkten Zugriffsdistanz und für den betrachteten Kandidatenpool ist die Autotuningphase maximal 19 Zeitschritte lang, für ODE-Systeme mit unbeschränkter Zugriffsdistanz müssen maximal 11 Autotuning-Schritte ausgeführt werden. Ein Aufwärmschritt wurde bei dieser Angabe der Anzahl der Autotuning-Schritte bereits mitgezählt.

8.5. Experimentelle Evaluierung des selbstadaptiven ODE-Solvers mit integrierter Blockgrößenauswahl

In diesem Abschnitt wird die Performance des selbstadaptiven ODE-Solvers auf drei unterschiedlichen Rechnersystemen untersucht. Die Beschreibung dieser Systeme ist im Anhang B gegeben. Für die Experimente wurde der GCC-Compiler in der Version 4.4.3 mit der Optimierungsstufe `-O3` und den Compilerflags `-D_REENTRANT -fPIC -std=c++0x -D_STDC_LIMIT_MACROS -g -rdynamic` genutzt.

8.5.1. Experimentelle Validierung modellbasierter Blockgrößenauswahl

Im Abschnitt 8.4.2 sind für jede Implementierungsvariante mehrere laufzeitrelevante Arbeitsräume identifiziert worden. Als erstes wird daher am Beispiel der Implementierungsvariante *PipeDb2mt* analysiert, welchen Einfluss diese auf die Laufzeit einer Implementierungsvariante ausüben. Außerdem wird untersucht, wie stark eine wachsende Blockgröße das Laufzeitverhalten einer Implementierungsvariante beeinflusst. Ferner wird die Qualität der vom Modell gewählten Blockgrößen bestimmt und der Zusammenhang zwischen effizienten Blockgrößen und der Anzahl von Cache-Fehlzugriffen analysiert.

Einfluss der Wahl einer Blockgröße auf die Laufzeit von Implementierungsvarianten

Die Abbildung 8.5 zeigt die Laufzeiten der Implementierungsvariante *PipeDb2mt* in Abhängigkeit von der Blockgröße für das Testproblem BRUSS2D auf zwei unterschiedlichen Rechnersystemen unter Verwendung der Problemgröße $n = 4.5 \cdot 10^6$ und des Verfahrens Lobatto IIIC (8) [HNW09b]. Die Blockgröße wurde – angefangen bei der Blockgröße 10 bis hin zur Blockgröße 5000 – mit einer Schrittweite von jeweils 50 erhöht, danach bis zur Blockgröße $1.0 \cdot 10^6$ mit einer Schrittweite von 500, ab der Blockgröße $1.0 \cdot 10^6$ wurden noch größere Schrittweiten verwendet.

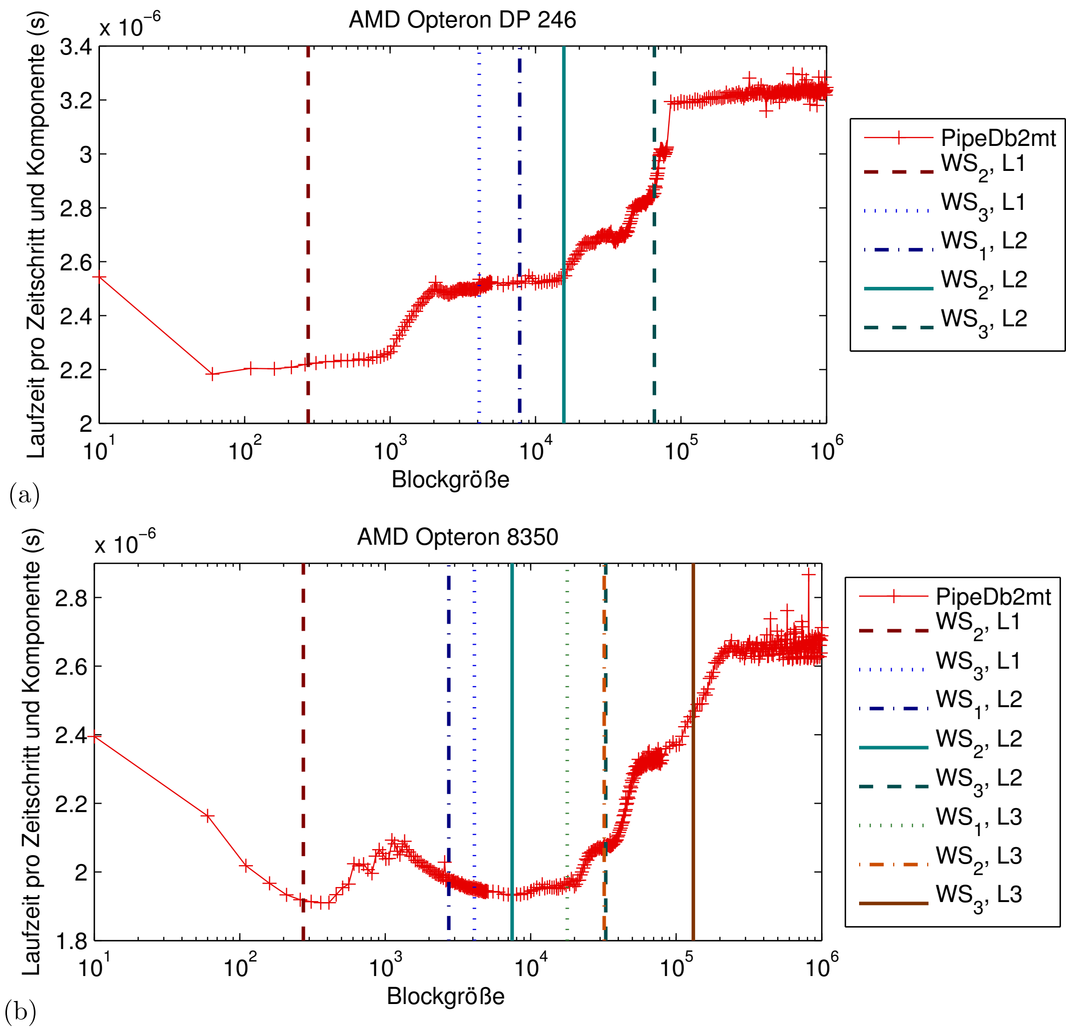


Abbildung 8.5.: Laufzeit der Implementierungsvariante *PipeDb2mt* in Abhängigkeit von der Blockgröße für das Testproblem BRUSS2D mit $n = 4.5 \cdot 10^6$ und dem Lobatto IIIC (8)-Verfahren auf (a) dem AMD Opteron 246 und (b) dem AMD Opteron 8350.

Auf dem AMD Opteron DP 246 (Abb. 8.5 (a)) zeigt die Laufzeitkurve der Implementierung *PipeDb2mt* ab der Blockgröße 60 einen kontinuierlichen Anstieg der Laufzeit, bis zur Blockgröße $\approx 7.5 \cdot 10^4$. Wächst die Blockgröße über diesen Wert hinaus, so bleibt die Laufzeit konstant. Der Grund für den kontinuierlichen Anstieg der Laufzeit ist, dass eine Erhöhung der Blockgröße zur einer Vergrößerung von Arbeitsräumen einer Implementierungsvariante führt. Ab einer bestimmten Blockgröße übersteigt die Größe laufzeitrelevanter Arbeitsräume die Kapazität des L1- oder L2-Caches. Die Arbeitsräume fallen nach und nach aus bestimmten Cache-Stufen heraus und dadurch steigt die Laufzeit der Implementierung stets an.

Für die Systemgröße $n = 4.5 \cdot 10^6$ passt weder der Arbeitsraum eines Zeitschrittes noch der des Korrektorschrittes in den L1- bzw. L2-Cache des AMD Opterons DP 246. Der kleinste Arbeitsraum der Implementierung *PipeDb2mt* ist $WS_3 = 2 \cdot ts$. Dieser überschreitet die Kapazität des L2-Caches ab einer Blockgröße von $\approx 6.6 \cdot 10^4$. Ist die Blockgröße bereits so groß, dass kein Arbeitsraum mehr in den L1- oder L2-Cache passt, so steigt die Laufzeit der Implementierung *PipeDb2mt* nicht mehr an. Für $n = 4.5 \cdot 10^6$ beträgt der maximale Performanceverlust 33 %, bezogen auf die optimale Blockgröße 60. In weiteren Experimenten (gestartet mit BRUSS2D, aber für andere Systemgrößen) ist für die Blockgrößen nahe an n ein Performanceverlust von über 50 % registriert worden. Das Modell der Blockgrößenauswahl würde auf diesem System zwei Blockgrößen vorschlagen: $ts_{min} = 128$ und $ts_{small} = 274$. Beide Blockgrößen liegen in einem guten Bereich des Suchraumes, der Performanceverlust zur optimalen Blockgröße ist bei ts_{min} $< 1\,\%$ und bei $ts_{small} < 2\,\%$.

Auch auf dem zweiten Rechnersystem AMD Opteron 8350 passt weder der Arbeitsraum eines Zeitschrittes noch der eines Korrektorschrittes bei einer Systemgröße von $n = 4.5 \cdot 10^6$ in die Cache-Hierarchie. Auf diesem Rechnersystem ist ein Anstieg der Laufzeit erst ab der Blockgröße ≈ 410 zu beobachten. Dies ist insofern interessant, als ab der Blockgröße 274 der Arbeitsraum WS_2, der einer Iteration der i-Schleife entspricht, die Größe des L1-Caches überschreitet. Der Arbeitsraum WS_2 ist der größte Arbeitsraum, der bis zur Blockgröße 274 vollständig in den L1-Cache passt. Ein weiterer Anstieg der Laufzeit erfolgt, wenn der Arbeitsraum WS_2 beginnt, aus dem L2-Cache herauszufallen. Wie bereits beim AMD Opteron DP 246 stabilisiert sich die Laufzeit erst, wenn der kleinste Arbeitsraum WS_3 aus der höchsten Cache-Stufe, hier L3, herausfällt. Für $n = 4.5 \cdot 10^6$ beträgt der maximale Performanceverlust auf diesem Rechner 40 %, bezogen auf die optimale Blockgröße 410. Weil der L1-Cache des AMD Opterons 8350 genauso groß ist wie der L1-Cache des AMD Opteron DP 246, schlägt das Blockgrößenauswahlmodell für beide Systeme gleiche Blockgrößen vor, nämlich $ts_{min} = 128$ und $ts_{small} = 274$. Beide Blockgrößen liegen in einem guten Bereich des Suchraumes, die optimale Blockgröße 410 ist $\approx 6\,\%$ schneller als die Blockgröße ts_{min} und nur $\approx 4\,\%$ schneller als die Blockgröße ts_{small}.

Bewertung der Qualität der modell-gewählten Blockgrößen

Die Abbildung 8.6 zeigt die Qualität der modell-gewählten Blockgrößen ts_{min} und ts_{small} bezogen auf die Laufzeit der optimalen Blockgröße für die Implementierungsvariante

PipeDb2mt auf drei unterschiedlichen Zielarchitekturen für das Testproblem BRUSS2D und das Lobatto IIIC (8)-Verfahren.

Je näher die Laufzeit der gewählten Blockgröße an der Laufzeit der optimalen Blockgröße ist, desto besser ist ihre Qualität. In den Experimenten wurde die Blockgröße mit der Schrittweite von 100 erhöht, bis die Blockgröße 5000 erreicht wurde.

In den Konturdiagrammen sind für unterschiedliche Systemgrößen n die relativen Laufzeiten der Implementierungsvariante *PipeDb2mt* gegenüber der besten über den gesamten Bereich der Blockgrößen $[1, 5000]$ erhaltenen Laufzeit zu sehen. Die Farbskalierung gibt die Qualität der Blockgrößen an. Die blauen Regionen entsprechen den Bereichen mit guter Wahl der Blockgröße, während die roten Regionen die Bereiche mit weniger gut geeigneten Blockgrößen repräsentieren.

Der Laufzeitunterschied zwischen einzelnen Blockgrößen aus dem Bereich $[1, 5000]$ beträgt auf dem AMD Opteron DP 246 $\lesssim 13\,\%$, auf dem AMD Opteron 8350 $\lesssim 11\,\%$ und auf dem Intel Core 2 Duo E8400 $\lesssim 4\,\%$. Bei sehr großen Blockgrößen jedoch, die nahe an n heranreichen, ist ein Performanceverlust von mehr als 40 % möglich (s. Abb. 8.5 (b)).

Auf dem AMD Opteron DP 246 System wäre die optimale Strategie die Wahl einer Blockgröße kleiner als ≈ 700, während auf dem AMD Opteron 8350 nicht nur kleine Blockgrößen aus dem Bereich $[200, 500]$ gute Performance liefern, sondern auch Blockgrößen aus dem Bereich $[3500, 5000]$. Beim Betrachten der Laufzeitergebnisse auf diesem System fällt auf, dass für manche Bereiche von Systemgrößen die Wahl einer Blockgröße aus dem Bereich $[1, 5000]$ keinen nennenswerten Einfluss auf die Laufzeit hat.

Das Intel Core 2 Duo E8400 System reagiert weniger sensibel auf die Wahl einer Blockgröße aus dem betrachteten Bereich. Dennoch sollten auch auf diesem System vorzugsweise die Blockgrößen kleiner als ≈ 2200 gewählt werden, um eine hohe Performance zu erzielen.

Generell lässt sich auf allen drei Zielarchitekturen beobachten, dass mindestens eine modell-gewählte Blockgröße in einem guten Bereich des Suchraumes liegt.

Zwei weitere Abbildungen 8.7 und 8.8 demonstrieren, dass die modellbasierte Vorauswahl von Blockgrößen auch für andere Implementierungen als *PipeDb2mt* und für andere Korrektorverfahren gut funktioniert. Die Abbildung 8.7 zeigt die Laufzeit der Implementierungen *PipeDb1mt* und *ppDb1mt* pro Zeitschritt und Komponente in Abhängigkeit der Blockgröße für $n = 4.5 \cdot 10^6$ $(N = 1500)$ auf dem AMD Opteron DP 6172 Rechnersystem, die Abbildung 8.8 auf dem AMD Opteron 8350. Es wurde erneut das Problem BRUSS2D verwendet. Die oberen zwei Abbildungen zeigen die Ergebnisse unter Verwendung des Korrektorverfahrens Lobatto IIIC (8), für die Experimente in den unteren Abbildungen wurde das Radau IA (5)-Verfahren benutzt. Die Blockgröße wurde beginnend mit dem Wert 3000 (dies entspricht der Zugriffsdistanz $d(\mathbf{f}) = 2N$ und der kleinstmöglichen Blockgröße für diese Implementierungsvarianten), bis zum Wert 24000 mit einer Schrittweite von 1000 erhöht. Danach wurde die Blockgröße verdoppelt, bis entweder die Systemgröße erreicht wurde oder bei der Implementierung *ppDb1mt* nicht genug Blöcke zum Aufbau einer vollen Pipeline verfügbar waren. Das analytische Modell liefert auch für die Implementierungen *PipeDb1mt* und *ppDb1mt* und unterschiedliche Korrektorverfahren gute Blockgrößen. Auf dem AMD Opteron DP 6172 ist

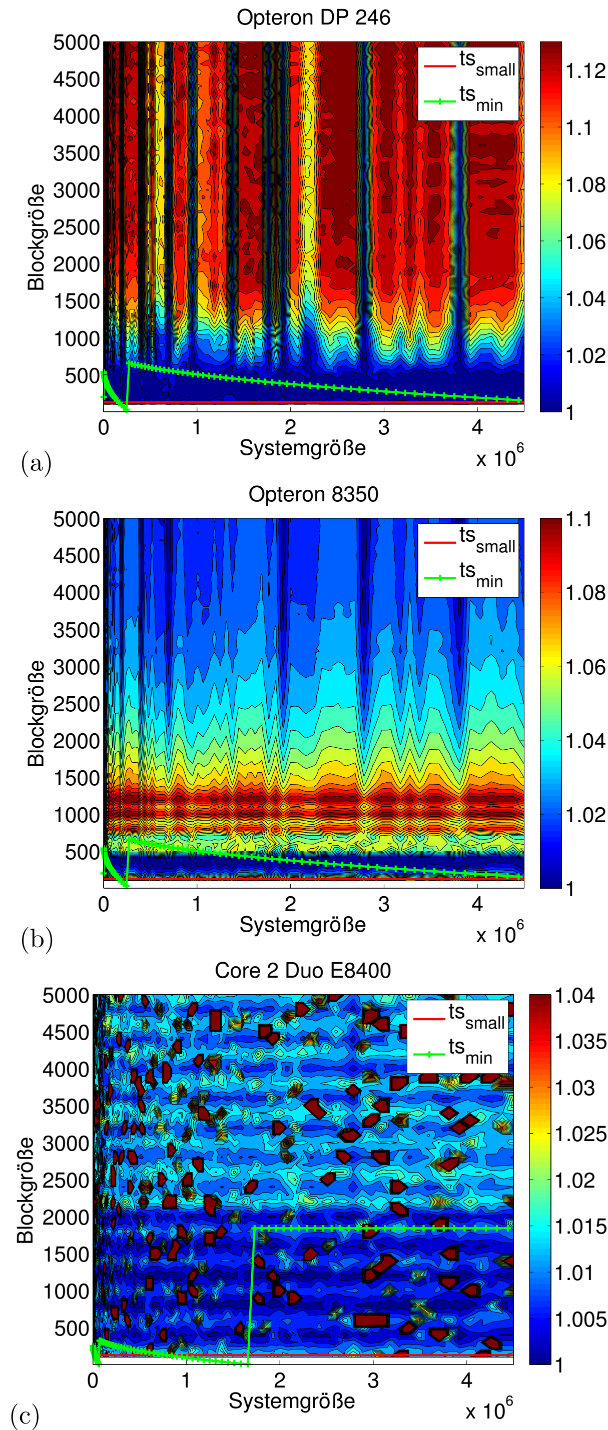


Abbildung 8.6.: Qualität der modell-gewählten Blockgrößen für die Implementierungsvariante *PipeDb2mt* und das Testproblem BRUSS2D. (a) Opteron DP 246, (b) Opteron 8350, (c) Core 2 Duo E8400.

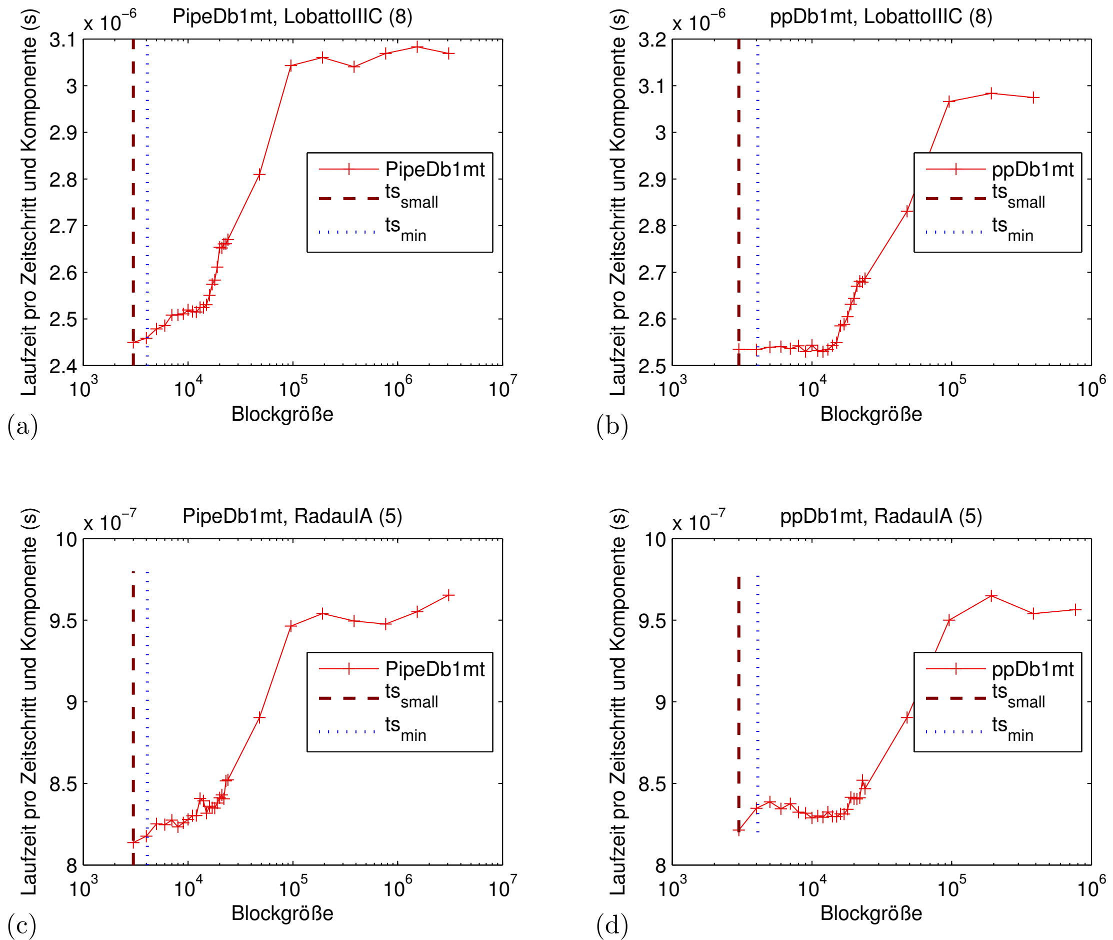


Abbildung 8.7.: Qualität der modell-gewählten Blockgrößen ts_{min} und ts_{small} für das Testproblem BRUSS2D mit $n = 4.5 \cdot 10^6$. Rechnersystem: AMD Opteron DP 6172. (a) *PipeDb1mt*, Lobatto IIIC (8), (b) *ppDb1mt*, Lobatto IIIC (8), (c) *PipeDb1mt*, Radau IA (5), (d) *ppDb1mt*, Radau IA (5).

die Blockgröße $ts_{small} = 3000$ die optimale Blockgröße in allen vier Testszenarien. Der Performanceunterschied der zweiten Blockgröße ts_{min} zu der optimalen Blockgröße liegt unter 1 %.

Auf dem zweiten Rechnersystem ist die Qualität der modell-gewählten Blockgrößen vergleichbar mit der auf dem AMD Opteron DP 6172. In Abb. 8.8 (a) ist die Laufzeit der Blockgröße ts_{small} maximal ≈ 2.5 % schlechter als die Laufzeit der optimalen Blockgröße 9000. Die Blockgröße ts_{min} ist lediglich ≈ 0.1 % schlechter als die optimale Blockgröße. Auch in anderen Experimenten (Abb. 8.8 (b), (c), (d)) ist eine ähnlich gute Qualität der modellbasierten Blockgrößen zu erkennen.

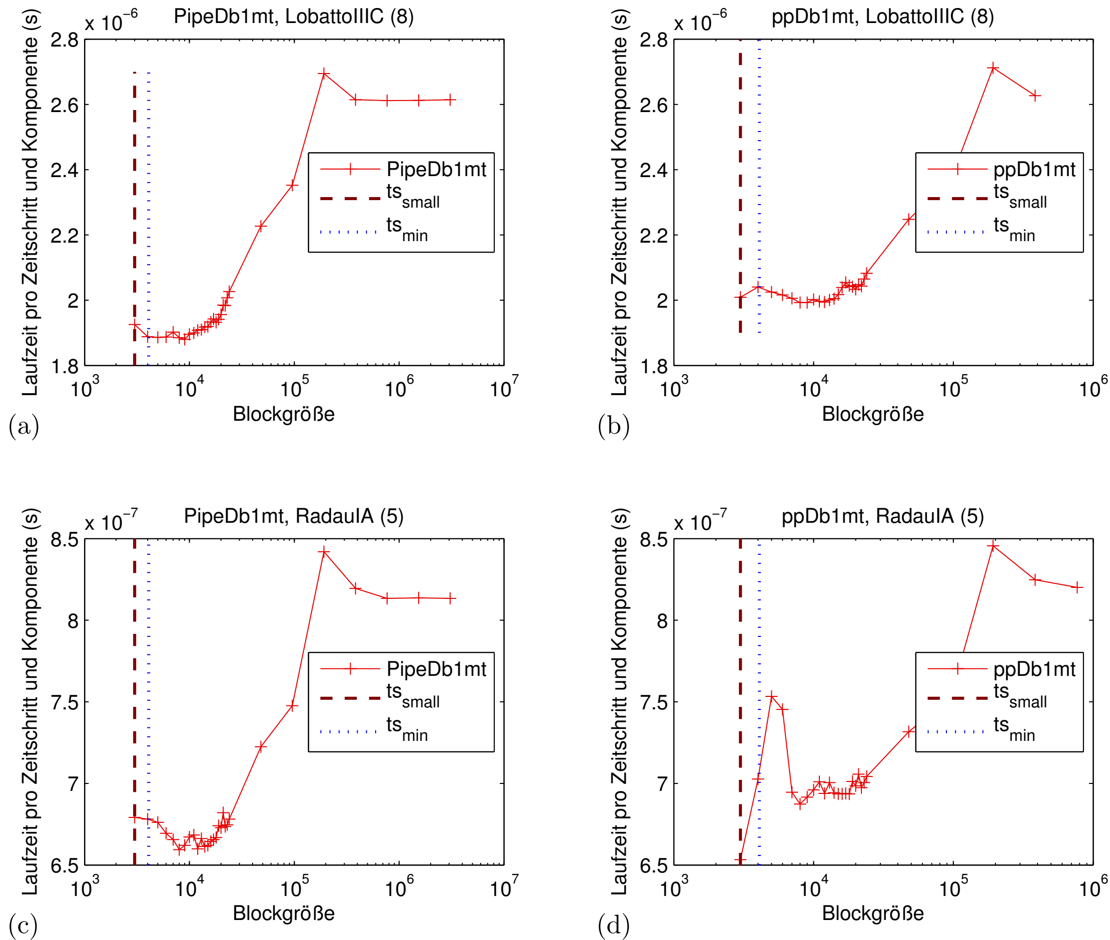


Abbildung 8.8.: Qualität der modell-gewählten Blockgrößen ts_{min} und ts_{small}. Testproblem: BRUSS2D mit $n = 4.5 \cdot 10^6$. Rechnersystem: AMD Opteron 8350. (a) *PipeDb1mt*, Lobatto IIIC (8), (b) *ppDb1mt*, Lobatto IIIC (8), (c) *PipeDb1mt*, Radau IA (5), (d) *ppDb1mt*, Radau IA (5).

Der Zusammenhang zwischen der Anzahl von Cache-Fehlzugriffen und effizienten Blockgrößen

Die Abbildung 8.9 zeigt die normierte Anzahl der L1- bzw. L2-Cache-Fehlzugriffe für unterschiedliche System- und Blockgrößen für die Implementierung *PipeDb2mt*, das Testproblem BRUSS2D und das Lobatto IIIC (8)-Verfahren auf dem AMD Opteron DP 246 System.

In den Konturdiagrammen sind für unterschiedliche Systemgrößen n die Cache-Fehlzugriffe gegen die minimale über den gesamten Bereich der Blockgrößen erhaltene Anzahl der Cache-Fehlzugriffe aufgetragen.

Mit wachsender Blockgröße steigt auch die Anzahl der L1-Cache-Fehlzugriffe stark an. Ein Anstieg der L2-Cache-Fehlzugriffe kann dagegen nicht beobachtet werden. Die Anzahl der L1-Cache-Fehlzugriffe ist für unterschiedliche Blockgrößen um mehrere Größenordnungen höher als die Anzahl der L2-Cache-Fehlzugriffe. Man sieht auch, dass es Bereiche von Systemgrößen gibt, bei denen die Anzahl der L1-Cache-Fehlzugriffe durchgehend hoch ist und die Anzahl der L2-Cache-Fehlzugriffe kleiner als bei übrigen Systemgrößen. Diese Bereiche entsprechen den Systemgrößen, bei denen die Wahl einer Blockgröße aus dem Bereich $[1, 5000]$ keinen nennenswerten Einfluss auf die Laufzeit hatte (vgl. Abb. 8.6 (a)).

Mit wachsender System- und Blockgröße übersteigt die Größe einzelner Arbeitsräume die Kapazität des L1-Caches. Dadurch beginnen die Teile dieser Arbeitsräume aus dem L1-Cache herauszufallen und die Anzahl der L1-Cache-Fehlzugriffe steigt an. Da die meisten Arbeitsräume aber immer noch in den L2-Cache passen, wenn die Blockgröße $\lesssim 5000$ gewählt wird, bleibt die Anzahl der L2-Cache-Fehlzugriffe bei steigender Blockgröße nahezu konstant.

Insgesamt bestätigen die Experimente, dass die Cachekapazität und die Größe laufzeitrelevanter Arbeitsräume einen großen Einfluss auf die Laufzeit einer Implementierungsvariante ausüben und deshalb wichtige Faktoren für die Blockgrößenauswahl sind. Außerdem zeigen die Experimente, dass es zum Erreichen einer hohen Performance ausreicht, die Blockgröße so zu wählen, dass die Größe laufzeitrelevanter Arbeitsräume die Kapazität der schnellsten Cache-Stufe nicht überschreitet.

8.5.2. Vergleich der Laufzeit nicht-adaptiver Implementierungsvarianten und des selbstadaptiven Solvers

Die Abbildung 8.10 zeigt einen Vergleich der Laufzeiten pro Zeitschritt und Komponente der nicht-adaptiven Implementierungsvarianten und des selbstadaptiven Solvers. Für die nicht-adaptiven Implementierungsvarianten mit Loop-Tiling wurde die Zugriffsdistanz $d(\mathbf{f})$ als Blockgröße verwendet, weil es einerseits die kleinstmögliche Blockgröße ist, die für die spezialisierten Varianten gewählt werden kann, und sie andererseits ausreichend groß für die Ausnutzung der räumlichen und zeitlichen Lokalität ist. Wie bei den in Kapitel 7.7 beschriebenen Experimenten wurde die Gesamtzahl auszuführender Zeitschritte begrenzt, der selbstadaptive Solver führt zwischen 60 und 850 Zeitschritte durch.

Beim Vergleich nicht-adaptiver Implementierungsvarianten auf dem Rechnersystem AMD Opteron DP 246 können drei Bereiche von Systemgrößen identifiziert werden, in denen sich die Laufzeitreihenfolge der Implementierungsvarianten unterscheidet. Für kleine Problemgrößen $n \lesssim 1.6 \cdot 10^4$ ist die schnellste Implementierungsvariante *E*. Für Problemgrößen $n \lesssim 0.6 \cdot 10^6$ sind *PipeDb2mt* oder *ppDb1mt* die schnellsten Varianten. Für noch größere Systemgrößen ist die Implementierung *E* erneut die effizienteste Variante. Die höchste Cache-Stufe ist bei diesem Rechner L2, diese hat eine Kapazität von 1 MB. Bei allgemeinen Implementierungsvarianten übersteigt die Größe des Arbeitsraumes eines Zeitschrittes die Kapazität des L2-Caches ab einer Systemgröße von $n \gtrsim 1.1 \cdot 10^4$, bei spezialisierten Implementierungsvarianten ab $n \gtrsim 1.5 \cdot 10^4$. Im Vergleich zu anderen

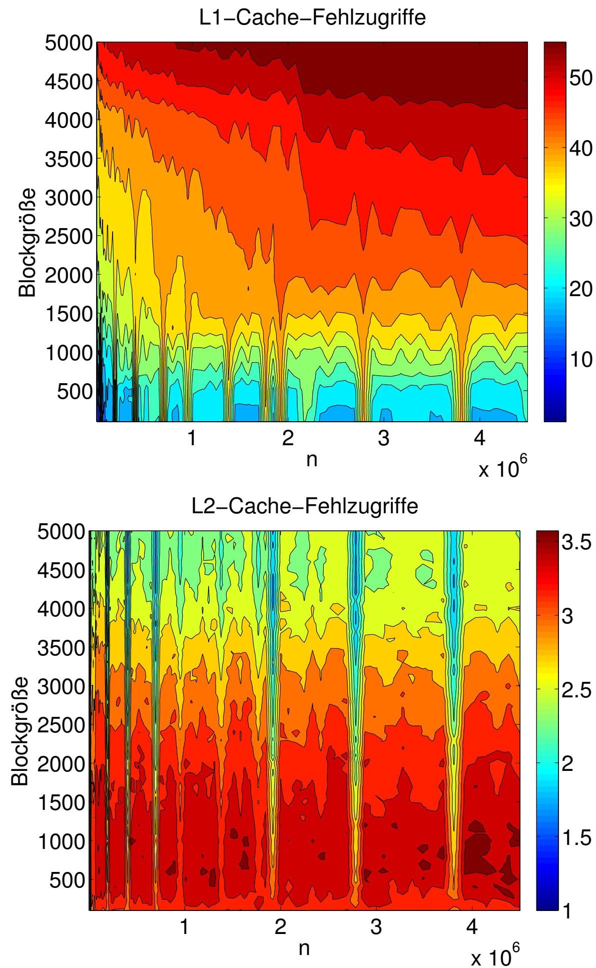


Abbildung 8.9.: Die normierte Anzahl der L1- und L2-Cache-Fehlzugriffe für unterschiedliche System- und Blockgrößen für die Implementierung *PipeDb2mt* und das Testproblem BRUSS2D auf dem AMD Opteron DP 246 System.

Implementierungen verwenden die Varianten *ppDb1mt* und *ppDb1m* das Konzept des Pipelinings zur verzögerten Berechnung der Korrektorschritte. Als Folge davon enthalten diese Varianten eine spezielle Arbeitsmenge WS_{PS} (s. Abschnitt 7.2.3). Diese besteht aus der Menge der Vektoren, die während eines diagonalen Laufs über die Korrektorschritte referenziert werden und ist von der Größenordnung $\Theta(smB)$ (s. Formel (7.1)). Da der Arbeitsraum eines Pipelining-Schrittes WS_{PS} nicht durch die Systemgröße n, sondern durch die Anzahl der Korrektorschritte m bestimmt wird, ist er wesentlich kleiner als der Arbeitsraum eines Zeitschrittes. Die Größe des Arbeitsraumes WS_{PS} übersteigt die Größe des L2-Caches ab der Systemgröße $n \gtrsim 0.6 \cdot 10^6$. Dies ist genau die Stelle, an der sich die schnellste Implementierungsvariante von *ppDb1mt* zu *E* ändert. Über den gesamten Bereich der Systemgrößen betrachtet erreicht der selbstadaptive Solver auf

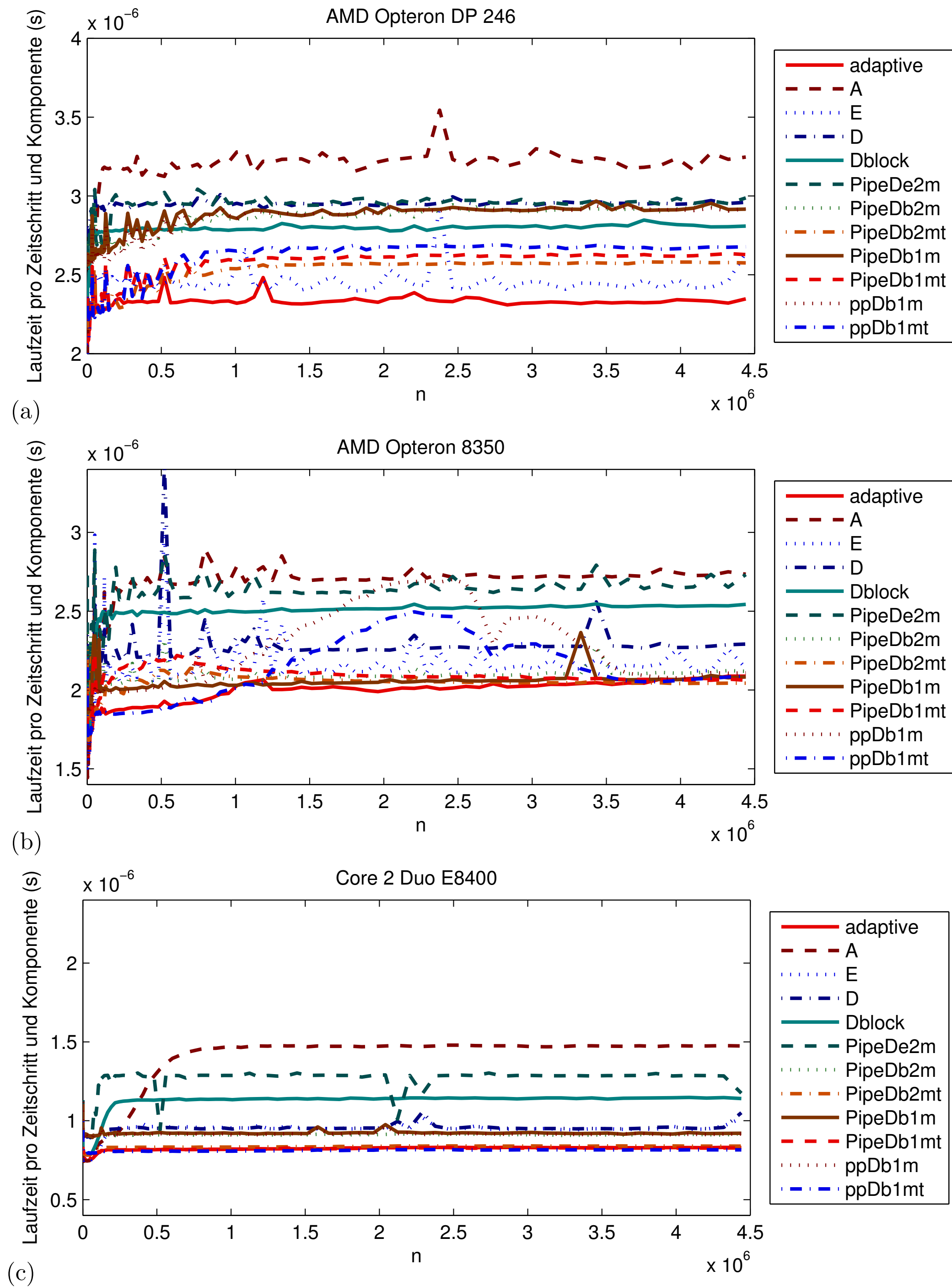


Abbildung 8.10.: Vergleich der normierten Laufzeiten in Abhängigkeit von der Systemgröße für das Testproblem BRUSS2D und das Lobatto IIIC (8)-Verfahren. (a) AMD Opteron DP 246, (b) AMD Opteron 8350, (c) Intel Core 2 Duo E8400.

dieser Plattform entweder eine ähnliche oder sogar eine bessere Performance als die jeweils beste Implementierungsvariante. Für $n \lesssim 2.3 \cdot 10^5$ erzielt der selbstadaptive Solver annähernd die gleiche Performance wie einer der schnellen Implementierungsvarianten in diesem Bereich: *E* und *ppDb1mt*. Für $n \gtrsim 2.3 \cdot 10^5$ liefert der selbstadaptive Solver sogar eine bessere Laufzeit als die schnellste nicht-adaptive Implementierungsvariante. Das passiert, weil der selbstadaptive Solver statt die Implementierung *E* oder *PipeDb2mt* mit vordefinierter Blockgröße $d(\mathbf{f})$ zu wählen, die Implementierung *PipeDb2mt* wählt, stattdessen aber mit einer effizienteren Blockgröße als $d(\mathbf{f})$.

Auch auf dem zweiten Rechnersystem AMD Opteron 8350 sind Bereiche zu sehen, an denen sich die Reihenfolge der Implementierungsvarianten ändert. Für Problemgrößen $n \lesssim 1.6 \cdot 10^4$ ist A die effizienteste Implementierung, für $1.6 \cdot 10^4 \lesssim n \lesssim 1.1 \cdot 10^6$ erzielt die Variante *ppDb1mt* die beste Performance. Für noch größere Problemgrößen ist *PipeDb2mt* die schnellste Variante. Der selbstadaptive Solver erreicht, wie schon auf dem AMD Opteron DP 246, entweder eine ähnliche Performance wie die schnellste nicht-adaptive Implementierung oder ist durch die Auswahl geeigneter Blockgrößen sogar schneller als diese.

Auf dem Intel Core 2 Duo E8400 System ist für $n \lesssim 2.0 \cdot 10^4$ die Implementierung A die schnellste Variante, bedingt dadurch, dass der Arbeitsraum eines Zeitschrittes komplett in den 6 MB großen L2-Cache passt. Im Bereich der Systemgrößen $2.0 \cdot 10^4 \lesssim n \lesssim 7.0 \cdot 10^4$ arbeitet die Implementierung *PipeDb2mt* am effizientesten. Ab der Systemgröße $n \gtrsim 7.0 \cdot 10^4$ liefert die Implementierung *ppDb1mt* die beste Laufzeit, weil der Arbeitsraum eines Pipelining-Schrittes für $n \approx 2.3 \cdot 10^7$ komplett in den L2-Cache passt. Wie erwartet, kann der selbstadaptive Solver auch auf dieser Plattform die beste Implementierungsvariante auswählen und das mit sehr geringem Zeitoverhead.

Insgesamt lässt sich festhalten, dass es keine nicht-adaptiven Implementierungsvariante gibt, die auf allen Rechnersystemen und für alle Problemgrößen die schnellste ist. Das Autotuning verursacht einen Overhead, der natürlich von der Gesamtanzahl der Zeitschritte zur Lösung eines Anfangswertproblems abhängt. Die Experimente in diesem Abschnitt haben jedoch gezeigt, dass der Autotuning-Overhead gering ist, selbst für die in den Experimenten verwendete relativ kleine Anzahl von Zeitschritten zwischen 60 und 850. Bei echten Simulationen hängt die Anzahl der Zeitschritte von dem zu lösenden Anfangswertproblem, dem verwendeten Korrektorverfahren, der vorgegebenen Genauigkeitstoleranz und der Länge des Integrationsintervalls ab. Die Gesamtanzahl der zur Lösung benötigten Zeitschritte kann daher im Bereich von mehreren Tausenden bis Millionen liegen. Die Länge des Integrationsintervalls und der maximal tolerierte Fehler für die Schrittweitenkontrolle werden vom Benutzer vorgegeben. Dieser möchte das AWP üblicherweise auf einem für ihn interessanten Bereich des Integrationsintervalls lösen, zum Beispiel bei BRUSS2D auf einem Bereich, auf dem eine chemische Reaktion das Gleichgewicht erreicht oder ein periodisches Verhalten zeigt. Der Benutzer entscheidet, wie lang er z. B. eine chemische Reaktion simuliert und mit welcher Genauigkeit.

Um dem Leser dennoch eine ungefähre Vorstellung davon zu geben, wie groß die Gesamtanzahl der Zeitschritte für die in dieser Arbeit betrachteten Probleme sein kann, wird das BRUSS2D-Problem betrachtet. Dieses ist ein Standardproblem, dessen Lösung

in [HNW09a] auf dem Integrationsintervall $[0; 11.5]$ gezeigt wurde. Unter Verwendung der Fehlertoleranz 10^{-6} und des Korrektorverfahrens Lobatto IIIC (8) braucht der ODE-Solver für dieses Integrationsintervall und die Systemgröße $n = 0.5 \cdot 10^6$ 11369 Zeitschritte. Für die Problemgröße $n = 8.0 \cdot 10^6$ werden 182313 Zeitschritte benötigt.

8.6. Zusammenfassung

In diesem Kapitel wurde der Autotuning-Grundalgorithmus aus Kapitel 7.5 um eine automatische Auswahl von Blockgrößen erweitert. Der resultierende selbstadaptive ODE-Solver bestimmt zunächst anhand eines Modells für jede im Kandidatenpool enthaltene Loop-Tiling-Variante eine kleine Menge potentiell effizienter Blockgrößen. Die Blockgrößen aus dieser Menge werden in der Autotuningphase auf der Zielarchitektur evaluiert und die beste Blockgröße wird anhand eines Vergleichs der Laufzeiten ermittelt.

Das analytische Modell basiert auf der Bestimmung von Arbeitsräumen der in der Implementierungsvariante auftretenden Schleifen und bezieht die relevanten Eigenschaften der Speicherhierarchie der Zielplattform in die Berechnung einer guten Blockgröße mit ein. Die Idee der Blockgrößenauswahl besteht darin, dass jeder Arbeitsraum, der einen großen Einfluss auf die Laufzeit ausübt, in der schnellstmöglichen Cache-Stufe in die er noch hineinpasst gespeichert werden soll. Zusätzlich verwendet das Modell eine auf praktischer Erfahrung basierende Heuristik. Für jede Implementierungsvariante werden maximal zwei Blockgrößen vorgeschlagen. Diese werden zur Laufzeit evaluiert. Da die Menge potentiell effizienter Blockgrößen schon vor Beginn der Berechnung feststeht und nur wenige Blockgrößen zur Laufzeit evaluiert werden, erfordert die automatische Auswahl von Blockgrößen nur geringen Zeitaufwand.

In Experimenten wurde zunächst untersucht, wie sich das Laufzeitverhalten einer Implementierungsvariante durch die wachsende Blockgröße verändert. Wird die Blockgröße erhöht, so wächst auch die Größe der Arbeitsräume. Ab einer bestimmten Blockgröße übersteigt die Größe eines Arbeitsraumes die Kapazität einer Cache-Stufe. Infolgedessen fällt dieser Arbeitsraum aus der entsprechenden Cache-Stufe heraus und die Laufzeit der Implementierung steigt an. Die Experimente haben gezeigt, dass, wenn die Größe eines Arbeitsraumes die Cachekapazität übersteigt, die Laufzeit nicht sprunghaft, sondern eher langsam ansteigt, weil zunächst noch Teile eines Arbeitsraumes in der Cache-Stufe verbleiben und dadurch wiederverwendet werden können. Des Weiteren wurde beobachtet, dass mit wachsender Blockgröße auch die Anzahl der L1-Cache-Fehlzugriffe ansteigt. Die Anzahl der L2-Cache-Fehlzugriffe bleibt dagegen weitgehend konstant. Dies hat die Ursache, dass die meisten Arbeitsräume immer noch in den L2-Cache passen, wenn die Blockgröße aus dem betrachten Bereich $[1, 5000]$ gewählt wird. Insgesamt betrachtet bestätigen die Experimente, dass die Cachekapazität und die Größe von Arbeitsräumen einen entscheidenden Einfluss auf die Laufzeit einer Implementierungsvariante ausüben.

Die Qualität modellbasierter Blockgrößen wurde systematisch auf drei verschiedenen Rechnersystemen und für unterschiedliche Implementierungsvarianten und Systemgrößen evaluiert. Die Ergebnisse haben gezeigt, dass das Modell präzise genug ist, um gute Blockgrößen vorschlagen zu können. In Experimenten lag mindestens eine der bestimm-

ten Blockgrößen in einem guten Bereich des Suchraumes. Auch die Effizienz der Implementierungsvarianten mit der zweiten Blockgröße war nur geringfügig schlechter als bei der Verwendung der optimalen Blockgröße. Generell haben die Experimente bestätigt, dass es zum Erreichen einer hohen Performance ausreichend ist, die Blockgröße so zu wählen, dass die laufzeitrelevanten Arbeitsräume einer Implementierung die Kapazität der schnellsten Cache-Stufe nicht überschreiten.

Der Autotuning-Overhead hängt generell von der Gesamtzahl der Zeitschritte zur Lösung eines Anfangswertproblems, der Anzahl der Implementierungsvarianten im Kandidatenpool und der Anzahl der zur Laufzeit evaluierten Blockgrößen ab. Die Laufzeitexperimente zeigten jedoch, dass für den betrachteten Kandidatenpool der Zeitaufwand zur Auswahl einer schnellen Implementierungsvariante mit guter Blockgröße gering ist. Das gilt selbst für eine relativ kleine Anzahl von Zeitschritten zwischen 60 und 850, die in Experimenten verwendet wurde. Bei echten Simulationen kann die Gesamtanzahl der Zeitschritte im Bereich von mehreren Hunderttausenden bis Millionen liegen; der Zeitaufwand zum Durchführen des Autotunings ist dann entsprechend klein.

9. Ein selbstadaptiver paralleler ODE-Solver für gemeinsamen Adressraum

Im Kapitel 3.5 wurden bereits verschiedene Konzepte zur parallelen Lösung von Anfangswertproblemen von ODEs vorgestellt. Die Verfahren zur numerischen Lösung von ODEs können bezüglich des Systems, der Methode oder der Zeit parallelisiert werden. Bei einigen speziellen ODE-Verfahren ist zudem eine Kombination verschiedener Arten der Parallelisierung möglich.

Im Abschnitt 3.5.1 wurde das Parallelitätspotential expliziter RK-Verfahren ausführlich diskutiert. PC-Verfahren vom RK-Typ der Form (3.27) bieten die Parallelität bezüglich der Methode, da ein Korrektorschritt s unabhängige Stufen enthält, die parallel berechnet werden können. Das Parallelitätspotential bezüglich der Methode ist allerdings durch die Anzahl der Stufen s des verwendeten Korrektorverfahrens beschränkt. Diese ist meistens ziemlich klein, z. B. ist beim Lobatto IIIC (8) $s = 5$. Somit reicht diese Art von Parallelisierung allein nicht aus, um die verfügbare Rechenleistung massiv-paralleler Computer optimal ausnutzen zu können.

Das größte Parallelitätspotential entfalten die expliziten PC-Verfahren vom RK-Typ bei der Ausnutzung der Parallelität bezüglich des Systems. Die obere Schranke für die maximale Anzahl zu nutzender Prozessoren ist durch die Dimension des Problems gegeben. Zum Lösen eines Systems der Dimension n können maximal n Prozessoren verwendet werden. Das Differentialgleichungssystem muss also groß genug sein, damit ein hoher Grad an Parallelität erreicht werden kann. Zudem würde bei Systemen mit kleiner Dimension höchstwahrscheinlich die Synchronisationszeit vorherrschen und somit die Effizienz der parallelen Ausführung begrenzen. Die Parallelisierung bezüglich des Systems lohnt sich daher in erster Linie für Anfangswertprobleme, die aus vielen Komponenten bestehen. Für kleine Systeme mit teuren Funktionsauswertungen f_j wäre jedoch eine Parallelisierung bezüglich des Systems, verbunden mit einer parallelen Berechnung einzelner Komponentenfunktionen, denkbar [Kor07].

Die Parallelisierung bezüglich der Zeitschritte ist bei expliziten RK-Verfahren nicht ohne Weiteres möglich, da diese zur Berechnung der Approximation $\boldsymbol{\eta}_{\kappa+1}$ im Schritt κ die Approximation $\boldsymbol{\eta}_{\kappa}$ des vorherigen Zeitschrittes $\kappa - 1$ brauchen. Für die in dieser Arbeit betrachteten Verfahren ist die Parallelisierung bezüglich der Zeitschritte aufgrund des verwendeten PC-Verfahrens jedoch möglich, wie in [vdHSvdV95] gezeigt wurde. Die Zeitschritte können mit Hilfe der Gauss-Seidel-Iteration überlappt werden, wenn die Iterationen vorangehender Zeitschritte anstatt der Approximation $\boldsymbol{\eta}_{\kappa}$ benutzt werden. Dadurch können insgesamt $m + N$ Iterationen parallel berechnet werden; m ist dabei die Anzahl der Korrektorschritte und N die Anzahl der Zeitschritte. Das resultierende Verfahren wurde als **PIRCAS GS** bezeichnet.

Für die Erstellung einer Auswahlmenge paralleler Implementierungskandidaten für ge-

meinsamen Adressraum werden Implementierungsvarianten verwendet, welche die Parallelität bezüglich des Systems nutzen (s. Abschnitt 9.2).

9.1. Programmierung mit gemeinsamen Variablen auf ccNUMA-Architektur

Die meisten modernen Parallelrechner mit gemeinsam genutztem Speicher besitzen eine ccNUMA-Speicher-Architektur. Bei dieser Art von Architektur verfügt jeder Prozessor über einen eigenen lokalen Speicher und kann über das Verbindungsnetzwerk auf ein Speichermodul der anderen Prozessoren zugreifen. Die Speicherzugriffszeiten sind abhängig vom Ort des Speichers und fallen damit unterschiedlich aus. Weil ein Speicherzugriff auf ein entferntes Modul über das Verbindungsnetzwerk erfolgt, ist dieser deutlich langsamer als ein Zugriff auf einen lokalen Speicher. Bei ccNUMA sorgt ein Cache-Kohärenz-Protokoll dafür, dass alle Prozessoren die aktuellen Daten verwenden [BU10]. Das SGI UV 2000 ist das aktuell größte ccNUMA-System, das in der Maximalkonfiguration bis zu 2048 Prozessorkerne enthalten kann [Cor14].

Eine Möglichkeit paralleler Programmierung auf Rechnern mit gemeinsamem Adressraum stellt die Thread-Programmierung dar. Diese basiert auf einem Modell, bei dem ein Prozess aus unabhängigen Folgen von Instruktionen (Kontrollflüssen) besteht, die parallel ausgeführt werden können. Als Prozess wird ein Programm bei seiner Ausführung bezeichnet [RR12a]. Die Kontrollflüsse eines Prozesses werden **Threads** genannt [RR12a]. Die Threads besitzen keinen eigenen Adressraum, sondern teilen sich den virtuellen Adressraum des zugehörigen Prozesses. Dadurch haben sie einen direkten Zugriff auf die Daten im virtuellen Adressraum und können diese über das Schreiben und Lesen von gemeinsamen Variablen austauschen. Somit sind zum Austausch von Daten keine expliziten Kommunikationsoperationen, aber dennoch Synchronisationsmechanismen, erforderlich.

Zur parallelen Programmierung mit gemeinsamen Variablen stellen die meisten Unix-artigen Betriebssysteme standardisierte APIs bereit. Beispiele dafür sind **Pthreads (POSIX-Threads)** [But97] für die Programmiersprache C bzw. C++ und **OpenMP (Open Multi Processing)** [ope14] für Programmierung mit C/C++ und Fortran.

Die Pthread-Bibliothek basiert auf dem von IEEE entwickelten Standard **POSIX (Portable Operating System Interface)**, nämlich auf dem Standard POSIX.1c. Da alle Threads auf einen gemeinsamen Speicherbereich Zugriff haben, konkurrieren diese um den Zugriff auf gemeinsam genutzte Datenstrukturen, was zu Problemen führen kann, wenn einer dieser Zugriffe schreibend erfolgt. Das kann dazu führen, dass das Ergebnis eines Programms von der Zugriffsreihenfolge der Threads auf gemeinsame Daten abhängt. Man spricht dabei von **zeitkritischen Abläufen (engl. race conditions)** [RR12a]. Zum Beispiel: Wenn der Thread T_i als erstes schreibend auf die globale Variable x zugreift und dann der Thread T_j lesend auf die selbe Variable, entsteht ein anderes Ergebnis, als wenn der Thread T_i die Variable x zuerst liest und dann der Thread T_j diese Variable verändert. Im ersten Fall überschreibt der Thread T_i die Variable x, bevor der Wert dieser Variablen vom Thread T_j gelesen wurde.

Die Teile eines Programms, bei denen nur ein Thread auf gemeinsame Variablen zugreifen kann, da ansonsten ein fehlerhafter Programmablauf entsteht, werden **kritische Bereiche (engl. critical sections)** genannt [RR12a]. Um einen korrekten Ablauf eines Programms gewährleisten zu können, muss der Zugriff auf kritische gemeinsame Variablen serialisiert werden. Pthread-Bibliotheken bieten dafür unterschiedliche Synchronisationsmechanismen an, u. a. können dazu **Mutexvariablen**, **Bedienungsvariablen**, **Lock-Mechanismen**, **Semaphore** und **Barrier** verwendet werden.

Damit eine parallele Implementierung auf einem ccNUMA-System bestmöglich skaliert, muss die Anzahl der Zugriffe auf entfernte Speichermodule so gering wie möglich gehalten werden. Um dies zu erreichen, müssen von einem Thread genutzte Datenstrukturen im lokalen Speicher des Prozessors auf dem der Thread ausgeführt wird, platziert werden. Dadurch kann der Thread schnell auf die Daten zugreifen und der Datentransfer durch das NUMA-Netzwerk wird reduziert. Das Linux-Betriebssystem nutzt zur Datenplatzierung die **First-Touch-Strategie** [HW10]. Bei dieser wird eine Speicherseite physikalisch im lokalen Speicher des Prozessors abgelegt, der als erstes diese Speicherseite beschreibt [HW10]. Da der Arbeitsspeicher aus gleich großen Speicherseiten besteht, die üblicherweise 4 kB groß sind, lässt es sich in der Regel nicht vermeiden, dass die Daten an den Grenzen der Datenverteilung im lokalen Speicher der Prozessoren von Nachbarthreads an den Rändern der Speicherseiten landen.

Die Voraussetzung für das effiziente Nutzen der First-Touch-Strategie ist, dass die Threads während der gesamten Laufzeit des Programms an die physikalischen Kerne gebunden sind. Der Vorgang des Bindens von Threads an die CPU-Kerne wird als **Thread- oder Prozess-Affinität** bezeichnet [Man10]. Generell versucht das Betriebssystem, die Arbeitslast optimal auf die verfügbaren Kerne zu verteilen und kann daher einen Thread von einem Kern auf einen anderen verschieben. Durch die Threadmigration geht die Lokalität der Speicherzugriffe verloren, da die Daten eines Threads danach u. U. im lokalen Speicher eines anderen Prozessors zu finden sind. Auch die Anzahl der Cache-Fehlzugriffe kann dadurch ansteigen, weil die Daten eines Threads im Cache eines anderen Prozessors, auf dem dieser Thread vor der Migration lief, abgelegt sind. Bei den meisten Betriebssystemen kann das Binden von Threads an die CPU-Kerne durch das Setzen der Bits in der Affinitäts-Bitmaske erfolgen. Die Maske enthält ein Bit für jede im System vorhandene CPU und kann über den Systemaufruf *sched_set_affinity()* vom Benutzer verändert werden.

Im nachfolgenden Abschnitt wird der Kandidatenpool paralleler Implementierungsvarianten für gemeinsamen Adressraum beschrieben, der eine Auswahlmenge von Varianten für den Autotuner bildet.

9.2. Kandidatenpool paralleler Implementierungsvarianten für gemeinsamen Adressraum

Parallele Implementierungsvarianten im Kandidatenpool basieren auf den Implementierungen aus [KR07b]. Im Rahmen dieser Dissertation wurde jedoch ihre Performance durch die Nutzung effizienter Synchronisationsoperationen entscheidend verbessert. Zu-

sätzlich wurden speziell für Rechner mit ccNUMA-Architektur neue Varianten erstellt, die den Argumentvektor $\mathbf{y}$ verteilt abspeichern.

Alle Implementierungsvarianten nutzen die Parallelität bezüglich des Systems aus. Die einzelnen Iterationen der Schleifen, die über die Systemdimension laufen, werden von mehreren Threads gleichzeitig abgearbeitet. Die Varianten sind mit POSIX-Threads und der Programmiersprache C implementiert worden. Zur Parallelisierung eines Programms müssen die Daten zwischen den zur Verfügung stehenden Prozessoren partitioniert werden. Dafür gibt es unterschiedliche Datenverteilungen, deren Beschreibung findet man in [RR00]. Für eindimensionale Felder können blockweise, zyklische und blockzyklische Verteilungen verwendet werden. Die Wahl einer Datenverteilung kann das Lokalitätsverhalten einer Implementierungsvariante beeinflussen. Außerdem kann eine ungünstige Datenverteilung zu einem Anstieg der Anzahl von Synchronisationsoperationen führen und somit eine gute Skalierbarkeit eines Programms verhindern.

Bei einer blockweisen Verteilung werden n Komponenten eines ODE-Systems zwischen p Threads aufgeteilt. Ist n ein ganzzahliges Vielfaches von p, so bekommt jeder Thread einen Block mit n/p aufeinander folgenden Komponenten zugeteilt. Andernfalls erhalten die ersten $n \bmod p$ Prozessoren jeweils $\lceil n/p \rceil$ Komponenten und alle übrigen Prozessoren $\lfloor n/p \rfloor$ Komponenten. Für das betrachtete PC-Verfahren ist eine blockweise Verteilung am besten geeignet. Diese weist jedem Thread ein Block zusammenhängender Komponenten zu und ermöglicht somit eine zeitliche und räumliche Wiederverwendung dieser innerhalb der Systemdimensionschleifen [KR07b]. Zudem ist eine blockweise Aufteilung insbesondere für Implementierungsvarianten mit beschränkter Zugriffsdistanz vorteilhaft. Besitzt ein ODE-System eine beschränkte Zugriffsdistanz, so werden für die Funktionsauswertung einer Komponente f_j nur $D = 2d(\mathbf{f})+1$ zusammenhängende Komponenten des Argumentvektors y benötigt. Gilt die Annahme $B \geq d(\mathbf{f})$ und wird eine blockweise Aufteilung verwendet, so braucht man maximal drei Blöcke $J-1$, J und $J+1$ für die Funktionsauswertungen der Komponenten im Block J. Ist der Block $J-1$ bzw. $J+1$ bei dem Thread selbst nicht vorhanden, so muss dieser höchstens von seinem linken bzw. rechten Nachbarn gelesen werden (siehe Abb. 9.3).

Die Tabellen 9.1 und 9.2 geben einen Überblick über die im Kandidatenpool enthaltenen parallelen Implementierungsvarianten. Als nächstes werden diese Varianten zum Verständnis der Funktionsweise des implementierten selbstadaptiven Solvers und den an Varianten durchgeführten Änderungen detailliert beschrieben.

9.2.1. Allgemeine Implementierungsvarianten

Bei den allgemeinen Implementierungsvarianten darf die Funktion der rechten Seite $\mathbf{f}(t, \mathbf{y})$ ein beliebiges Zugriffsmuster aufweisen (siehe dazu Abschnitt 7.2.2). Die Berechnung der Korrektorschritte muss sequentiell erfolgen, d. h. die über die Korrektorschritte iterierende Schleife k bildet die äußerste Schleife innerhalb der Zeitschrittfunktion. Parallele allgemeine Implementierungsvarianten verwenden die gleichen Datenstrukturen wie die entsprechenden sequentiellen Varianten. Jeder Thread kann auf gemeinsam genutzte Datenstrukturen zugreifen. Die einzelnen Korrektorschritte werden je nach Struktur der Implementierung durch eine oder zwei Barrier-Operationen getrennt. Eine Barri-

Implementierung	Datenstrukturen	Schleifenstruktur	Eigenschaften
A	$p \times \mathbf{F}_1^{(k)}, \ldots, \mathbf{F}_s^{(k)} \in \mathbb{R}^{n/p}$, $p \times \mathbf{F}_1^{(k-1)}, \ldots, \mathbf{F}_s^{(k-1)} \in \mathbb{R}^{n/p}$, $\mathbf{Y} \in \mathbb{R}^n$	k–l–i–j	vektororientiert: innere Schleifen iterieren über die Systemdimension; große räumliche Lokalität
E	$p \times \mathbf{F}_1^{(k)}, \ldots, \mathbf{F}_s^{(k)} \in \mathbb{R}^{n/p}$, $p \times \mathbf{F}_1^{(k-1)}, \ldots, \mathbf{F}_s^{(k-1)} \in \mathbb{R}^{n/p}$, $\mathbf{Y} \in \mathbb{R}^n$	k–l–j–i	nutzt die zeitlichen Lokalität der i Schleife bezüglich Schreibzugriffe auf die Argumentvektoren
EAblock	$p \times \mathbf{F}_1^{(k)}, \ldots, \mathbf{F}_s^{(k)} \in \mathbb{R}^{n/p}$, $p \times \mathbf{Y}_{\text{local}} \in \mathbb{R}^{n/p}$ $p \times \mathbf{F}_1^{(k-1)}, \ldots, \mathbf{F}_s^{(k-1)} \in \mathbb{R}^{n/p}$, $\mathbf{Y} \in \mathbb{R}^n$	k–l–j–i–jj	ähnlich zu Implementierung *E*, aber mit Loop-Tiling der j-Schleife mit der i-Schleife
D	$\mathbf{Y}_1^{(k)}, \ldots, \mathbf{Y}_s^{(k)} \in \mathbb{R}^n$, $\mathbf{Y}_1^{(k-1)}, \ldots, \mathbf{Y}_s^{(k-1)} \in \mathbb{R}^n$	k–i–j–l	nutzt die zeitlichen Lokalität der l-Schleife hinsichtlich der Lesezugriffe auf die Ergebnisse von Funktionsauswertungen
Dblock	$\mathbf{Y}_1^{(k)}, \ldots, \mathbf{Y}_s^{(k)} \in \mathbb{R}^n$, $\mathbf{Y}_1^{(k-1)}, \ldots, \mathbf{Y}_s^{(k-1)} \in \mathbb{R}^n$, $p \times \mathbf{F} \in \mathbb{R}^B$	k–i–j–l–jj	ähnlich zu *D*, aber mit Loop-Tiling der j-Schleife und Vertauschen mit der l-Schleife
PipeDe2m	$\mathbf{Y}_1^{(k)}, \ldots, \mathbf{Y}_s^{(k)} \in \mathbb{R}^n$, $\mathbf{Y}_1^{(k-1)}, \ldots, \mathbf{Y}_s^{(k-1)} \in \mathbb{R}^n$	k–j–i–l	basiert auf *D*; j-Schleife iteriert über die Schleifen l und i; nutzt zeitliche Lokalität der Schleifen i und l
PipeDb2m	$\mathbf{Y}_1^{(k)}, \ldots, \mathbf{Y}_s^{(k)} \in \mathbb{R}^n$, $\mathbf{Y}_1^{(k-1)}, \ldots, \mathbf{Y}_s^{(k-1)} \in \mathbb{R}^n$	k–j–i–jj–l	ähnlich zu *PipeDe2m*, aber zusätzlich Loop-Tiling der j-Schleife und Vertauschen mit der i-Schleife
PipeDb2mt	$\mathbf{Y}_1^{(k)}, \ldots, \mathbf{Y}_s^{(k)} \in \mathbb{R}^n$, $\mathbf{Y}_1^{(k-1)}, \ldots, \mathbf{Y}_s^{(k-1)} \in \mathbb{R}^n$, $p \times \mathbf{F} \in \mathbb{R}^B$	k–j–i–(jj)–l–jj	wie *PipeDb2m*, jedoch ist das Loop-Tiling auf die l-Schleife ausgeweitet worden

Tabelle 9.1.: Kandidatenpool allgemeiner paralleler Implementierungsvarianten für gemeinsamen Adressraum.

Implementierung	Datenstrukturen	Schleifenstruktur	Eigenschaften
PipeDb1m	$p \times \bar{Y} \in \mathbb{R}^{s,(m-1)\cdot(B+d)+2d+n/p}$	k–j–i–jj–l	ähnlich zu *PipeDb2m*, aber zur Minimierung des Speicherplatzbedarfs sind die Vektoren $\mathbf{Y}_l^{(k)}$ überlappt worden
PipeDb1mt	$p \times \bar{Y} \in \mathbb{R}^{s,(m-1)\cdot(B+d)+2d+n/p}$, $p \times \mathbf{F} \in \mathbb{R}^B$	k–j–i–(jj)–l–jj	beruht auf *PipeDb1m*, aber das Loop-Tiling ist auf die l-Schleife ausgeweitet worden
ppDb1m	$p \times \bar{Y} \in \mathbb{R}^{s,(m-1)\cdot(b+d)+2b+n/p}$	j–k–i–jj–l	basiert auf *PipeDb1m*; j- und k-Schleifen sind durch den Pipelining-Ansatz vertauscht
ppDb1mt	$p \times \bar{Y} \in \mathbb{R}^{s,(m-1)\cdot(b+d)+2b+n/p}$, $p \times \mathbf{F} \in \mathbb{R}^b$	j–k–i–(jj)–l–jj	beruht auf *ppDb1m*, allerdings ist das Loop-Tiling auf die l-Schleife ausgeweitet worden
PipeDb1myl	$p \times \bar{Y} \in \mathbb{R}^{s,(m-1)\cdot(B+d)+2d+n/p}$, $p \times \mathbf{y}_{\text{local}} \in \mathbb{R}^{2d+n/p}$	k–j–i–jj–l	ähnlich zu *PipeDb1m*, jedoch mit verteilter Speicherung des Argumentvektors $\mathbf{y}$
PipeDb1mtyl	$p \times \bar{Y} \in \mathbb{R}^{s,(m-1)\cdot(B+d)+2d+n/p}$, $p \times \mathbf{y}_{\text{local}} \in \mathbb{R}^{2d+n/p}$, $p \times \mathbf{F} \in \mathbb{R}^B$	k–j–i–(jj)–l–jj	ähnlich zu *PipeDb1mt*, jedoch mit verteilter Speicherung des Argumentvektors $\mathbf{y}$
ppDb1myl	$p \times \bar{Y} \in \mathbb{R}^{s,(m-1)\cdot(b+d)+2b+n/p}$, $p \times \mathbf{y}_{\text{local}} \in \mathbb{R}^{2d+n/p}$	j–k–i–jj–l	ähnlich zu *ppDb1m*, jedoch mit verteilter Speicherung des Argumentvektors $\mathbf{y}$
ppDb1mtyl	$p \times \bar{Y} \in \mathbb{R}^{s,(m-1)\cdot(b+d)+2b+n/p}$, $p \times \mathbf{y}_{\text{local}} \in \mathbb{R}^{2d+n/p}$, $p \times \mathbf{F} \in \mathbb{R}^b$	j–k–i–(jj)–l–jj	ähnlich zu *ppDb1mt*, jedoch mit verteilter Speicherung des Argumentvektors $\mathbf{y}$

Tabelle 9.2.: Aktueller Kandidatenpool spezialisierter paralleler Implementierungsvarianten für gemeinsamen Adressraum ($d = d(\mathbf{f})$, $b = \max(d(\mathbf{f}), B)$).

er stellt sicher, dass kein Thread mit der Programmberechnung fortfahren kann, bis alle Threads diese Operation aufgerufen haben. In den von der Implementierung *D* abgeleiteten Varianten führen die Threads eine Barrier-Operation in jedem Korrektorschritt aus. Dadurch wird gewährleistet, dass die Berechnung der Argumentvektoren $Y^{(\mathrm{prev})} = \left(\mathbf{Y}_1^{(k-1)}, \ldots, \mathbf{Y}_s^{(k-1)}\right)$ im vorherigen Korrektorschritt abgeschlossen ist, bevor diese für die Funktionsauswertungen und Berechnung von $Y^{(\mathrm{cur})} = \left(\mathbf{Y}_1^{(k)}, \ldots, \mathbf{Y}_s^{(k)}\right)$ im nachfolgenden Korrektorschritt benutzt werden. Die Abbildung 9.1 zeigt den Pseudocode eines Zeitschrittes der Implementierung *PipeDb2mt* als Beispiel für eine parallele Implementierungsvariante für gemeinsamen Adressraum.

Die Implementierungsvarianten *A*, *E* und *EAmt* verwenden einen einzigen Vektor $\mathbf{Y}$, um im aktuellen Korrektorschritt alle Funktionsargumente $F^{(\mathrm{cur})}$ anhand von $F^{(\mathrm{prev})}$ auszurechnen. Damit der Argumentvektor $\mathbf{Y}$ im nächsten Korrektorschritt nicht überschrieben wird, bevor die Berechnung von $F^{(\mathrm{prev})} = \left(\mathbf{F}_1^{(k-1)}, \ldots, \mathbf{F}_s^{(k-1)}\right)$ im vorherigen Korrektorschritt abgeschlossen ist, nutzen diese Varianten eine zweite Barrier-Operation am Ende jedes Korrektorschrittes.

Am Ende eines Zeitschrittes findet die Schrittweitenkontrolle statt. In dieser wird der lokale Fehler ϵ bestimmt und die neue Schrittweite berechnet. Anhand des lokalen Fehlers wird dann entschieden, ob der aktuelle Zeitschritt verworfen oder akzeptiert wird. Da die Berechnung der Zeitschritte nacheinander erfolgt, führen alle Threads den gleichen Zeitschritt mit der identischen Schrittweite aus. Das wiederum bedeutet, dass die Threads solange mit dem nächsten Schritt nicht fortfahren können, bis eine einheitliche Entscheidung über die Wahl einer Schrittweite und darüber, ob der Zeitschritt verworfen oder akzeptiert wird, vorliegt. Die Berechnungen innerhalb eines Zeitschrittes laufen parallel ab und so berechnet jeder Thread zunächst seinen lokalen Fehler. Innerhalb einer Barrier-Operation wird der lokale Fehler aller Threads verglichen und das Maximum bestimmt.

Beim Anlegen von Datenstrukturen muss die physikalisch verteilte Speicherstruktur moderner cc-NUMA-Systeme berücksichtigt werden. Um die Anzahl entfernter Speicherzugriffe möglichst gering zu halten, sollten die Vektoren eigentlich verteilt gespeichert werden, sodass die von einem Thread genutzten Vektorkomponenten auch in seinem lokalen Speicher platziert werden. Weil aber bei allgemeinen Implementierungsvarianten ein beliebiges Zugriffsmuster der Funktion $\mathbf{f}(t, \mathbf{y})$ erlaubt ist, so dass im schlimmsten Fall zur Berechnung einer Komponente $\mathbf{f}_j(t, \mathbf{y})$ alle Komponenten von $\mathbf{y}$ gebraucht werden, müssen alle Argumentvektoren für die Berechnung der Funktionsauswertungen als gemeinsame Datenstrukturen implementiert werden. Um die First-Touch-Strategie nutzen zu können, müssen die Datenstrukturen von den Threads in der gleichen Reihenfolge initialisiert werden, wie diese auch im Laufe des Programms benutzt werden.

Die in Tabelle 9.1 dargestellten allgemeinen Varianten wurden mit der Ausnahme der Implementierung *EAmt* in [KR07b] veröffentlicht. Im Rahmen dieser Arbeit wurden diese jedoch durch eine effiziente Implementierung einer Barrier-Operation maßgeblich verbessert. Die ursprünglich in [KR07b] verwendete Barrier aus der Pthread-Bibliothek wurde durch eine **warteschlangen-basierte (engl. queue-based)** Implementierung ersetzt,

```
    // m Korrektorschritte und Berechnung von Δη^(m−1) = Σ_{i=1}^{s} b_i F_i^(m−1)
 1: for (k = 0; k < m; k++)
 2: {
 3:     barrier();
 4:     swap(Y^(cur), Y^(prev));
 5:     for (j = first_comp; j <= last_comp − B + 1; j += B)
 6:     {
 7:         for (jj = 0; jj < B; jj++)
 8:             F_jj = h f_{j+jj} (t + c_0 h, (k == 0 ? η : Y_0^(prev)));
 9:         for (i = 1; i < s; i++)
10:             for (jj = 0; jj < B; jj++) Y_{i,j+jj}^(cur) = a_{i0} F_jj;
11:         if (k == m − 1)
12:             for (jj = 0; jj < B; jj++) Δη_{j+jj−first_comp}^(m−1) = b_0 F_jj;
13:         for (l = 0; l < s; l++)
14:         {
15:             for (jj = 0; jj < B; jj++)
16:                 F_jj = h f_{j+jj} (t + c_l h, (k == 0 ? η : Y_l^(prev)));
17:             for (i = 1; i < s; i++)
18:                 for (jj = 0; jj < B; jj++)
19:                 {
20:                     Y_{i,j+jj}^(cur) += a_{il} F_jj;
21:                     Y_{i,j+jj}^(cur) = Y_{i,j+jj}^(cur) + η_{j+jj};
22:                 }
23:             if (k == m − 1)
24:                 for (jj = 0; jj < B; jj++) Δη_{j+jj}^(m−1) += b_l F_jj;
25:         }
26:     }
27: }

    // Berechnung von Δη^(m) = Σ_{i=1}^{s} b_i F_i^(m)
28: barrier();
29: SWAP(Y^(cur), Y^(prev));
30: for (j = first_comp; j < last_comp − B + 1; j += B)
31: {
32:     for (jj = 0; jj < B; jj++)
33:         Δη_{j+jj−first_comp}^(m) = b_0 h f_{j+jj} (t + c_0 h, Y_0^(prev));
34:     for (l = 0; l < s; l++)
35:         for (jj = 0; jj < B; jj++)
36:             Δη_{j+jj−first_comp}^(m) += b_l h f_{j+jj} (t + c_l h, Y_l^(prev));
37: }

    // Berechnung von e = η_{κ+1} − η̂_{κ+1} und η_{κ+1} durch die Aktualisierung von η
38: for (j = first_comp; j < last_comp; j++)
39: {
40:     e_j = Δη_{j−first_comp}^(m) − Δη_{j−first_comp}^(m−1);
41:     η_j += Δη_{j−first_comp}^(m);
42: }
```

Abbildung 9.1.: Pseudocode eines Zeitschrittes der parallelen Implementierungsvariante *PipeDb2mt* für gemeinsamen Adressraum.

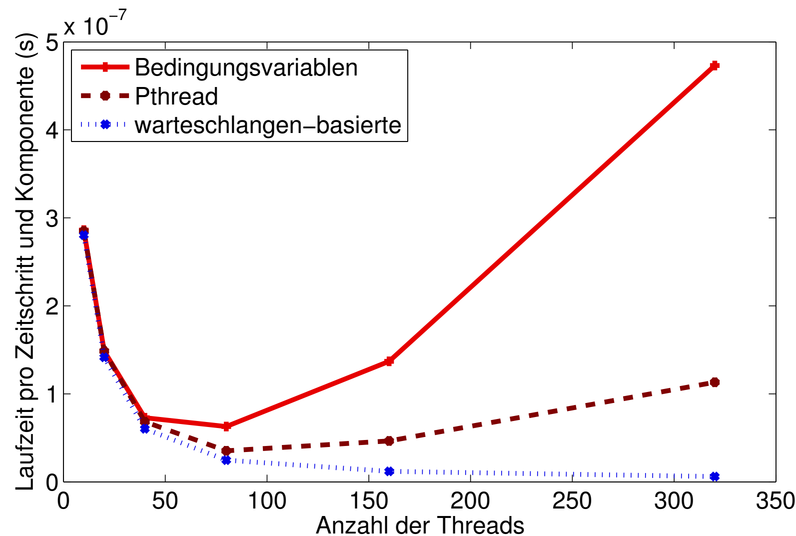


Abbildung 9.2.: Einfluss verschiedener Barrier-Implementierungen auf die Performance der allgemeinen Implementierungsvariante A. In dieser Abbildung werden verglichen: die von der Pthread-Bibliothek bereitgestellte (`pthread_barrier_wait()`) Barrier, eine auf Bedingungsvariablen aufbauende Barrier und eine warteschlangen-basierte, auf aktivem Warten beruhende Barrier [CC05]. Testplattform: SGI UV. Testproblem: BRUSS2D mit $n = 8.0 \cdot 10^6$. Korrektorverfahren: Lobatto III C(8).

die auf **aktivem Warten (engl. busy waiting)** beruht [CC05, CW08]. Die Gesamtzeit für eine Barrier-Operation setzt sich aus der Zeit für **Gather-** (jeder Thread signalisiert seine Ankunft an der Barrier) und **Release-** (die Threads verlassen die Barrier) **Phasen** zusammen. Bei einer warteschlangen-basierten Barrier verwaltet ein Koordinator (Master-Thread) ein globales Array von Flags. Dieses wird dazu benutzt, um die Ankunft der Threads zu dokumentieren, wie auch zum Signalisieren, wann die Threads die Barrier verlassen können. Während der Gather-Phase meldet jeder Thread seine Ankunft an der Barrier, indem er seine private Flag-Variable im globalen Array um eins inkrementiert. Gleichzeitig überprüft er, ob alle Threads an der Barrier bereits angekommen sind. Der Master-Thread prüft kontinuierlich nach, ob alle Threads die Barrier erreicht haben und die Release-Phase eingeleitet werden soll. Mehr Details zur Implementierung einer warteschlangen-basierten Barrier ist in [CC05, CW08] zu finden. Zur Minimierung von Speicherkonflikten wurden die Adressen der Elemente im globalen Array (Flags) mit Hilfe des Array-Paddings so ausgerichtet, dass diese in unterschiedliche Cachezeilen platziert werden.

Die Abbildung 9.2 veranschaulicht anhand der Variante A den Einfluss einer Barrier-Implementierung auf die Performance einer parallelen Implementierung. Wird bei einer großen Threadanzahl eine warteschlangen-basierte Barrier verwendet, so steigt die Performance der Implementierung A um mehr als 100 % im Vergleich zur Nutzung einer

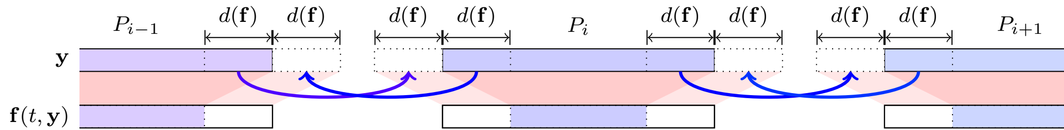


Abbildung 9.3.: Ein optimiertes Kommunikationsmuster für ODE-Systeme mit einer beschränkten Zugriffsdistanz.

Barrier aus der Pthread-Bibliothek.

9.2.2. Spezialisierte Implementierungsvarianten

Wie bereits im Abschnitt 7.2.3 erörtert, ist es für ODE-Probleme mit beschränkter Zugriffsdistanz möglich, zusätzliche auf die Lokalität hin optimierte Implementierungsvarianten zu erzeugen.

Durch die Ausnutzung einer beschränkten Zugriffsdistanz können die Argumentvektoren überlappt werden. Zudem ist eine verzögerte Berechnung der Korrektorschritte gemäß dem Pipelining-Ansatz realisierbar. Bei parallelen Implementierungsvarianten für gemeinsamen Adressraum bietet eine beschränkte Zugriffsdistanz zusätzlich die Möglichkeit, die Argumentvektoren für die Funktionsauswertungen verteilt zu speichern. Ferner können effizientere Mechanismen zur Synchronisation der Threads verwendet werden, da zum Berechnen einer Funktionskomponente $\mathbf{f}_j$ nicht mehr alle Komponenten n des Argumentvektors $\mathbf{y}$ gebraucht werden, sondern nur n/p Komponenten von $\mathbf{y}$ und zusätzlich jeweils $d(\mathbf{f})$ Komponenten, die vom rechten und vom linken Nachbarthread kopiert werden müssen (s. Abb. 9.3).

Weil die Daten nur zwischen den Nachbarthreads ausgetauscht werden, sind globale Barrier-Operationen nicht mehr nötig. Stattdessen reicht es aus, wenn ein **Lock-Mechanismus** zur Synchronisation benutzt wird. Dieser gewährleistet, dass nur ein Thread den kritischen Bereich betreten darf, und schützt somit den Zugriff auf gemeinsame Daten. Ein **Lock (auch Sperre genannt)** ist ein Datenobjekt eines speziellen Datentyps, auf das mittels zwei Funktionen: *lock()* und *unlock()* zugegriffen werden kann. Zu einem Zeitpunkt kann nur ein einziger Thread durch den Aufruf der Funktion *lock()* die Sperre erwerben und auf Daten im kritischen Programmsegment zugreifen. Alle anderen Threads können den kritischen Bereich solange nicht betreten, bis die Sperre mit Hilfe der Funktion *unlock()* von dem erwerbenden Thread wieder freigegeben wird.

Lock-Mechanismen können nach Art des Wartens auf die Freigabe einer Sperre in zwei Kategorien unterteilt werden: **aktiv-wartend (engl. busy-waiting)** und **passiv-wartend**. **Spin-Locks** sind aktiv-wartende Locks, die fortwährend Abfragen **(engl. Polling)** stellen, ob eine Sperre bereits freigegeben worden ist. Bei passiv-wartenden Locks werden die Threads schlafen gelegt, bis die Sperre verfügbar ist.

Die Pthread-Bibliothek verfügt über standardisierte Typen und Funktionen zur Synchronisation von Threads mittels Lock-Mechanismen. Mit einem Pthread-Mutex steht ein passiv wartender Lock-Mechanismus zur Verfügung. Dieser wird durch die Variable

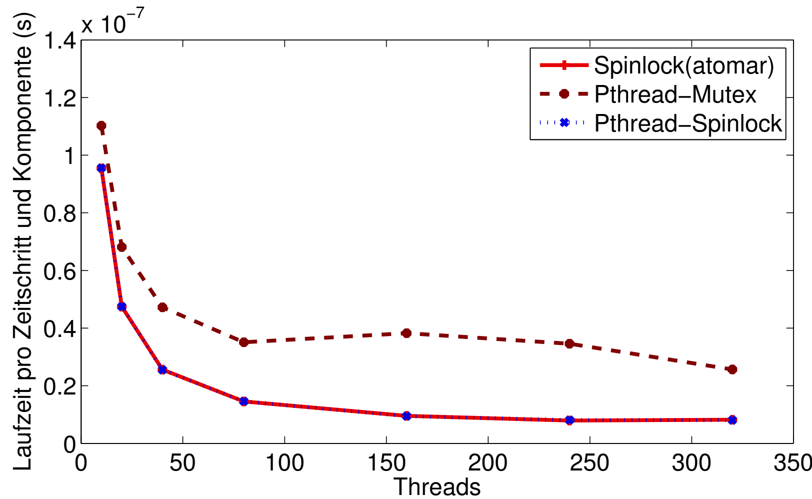


Abbildung 9.4.: Einfluss unterschiedlicher Lockmechanismen auf die Laufzeit der Implementierungsvariante *PipeDb1mt*. In der Abbildung werden Pthread-Mutex (passiv-wartend), Pthread-Spinlock (aktiv-wartend) und ein Spinlock auf Basis atomarer Operationen (aktiv-wartend) miteinander verglichen. Testplattform: SGI UV. Testproblem: BRUSS2D mit $n = 0.5 \cdot 10^6$. Korrektorverfahren: Lobatto III C(8).

pthread_mutex_t und die Funktionen mit Präfix *pthread_mutex_* repräsentiert. Außerdem stehen Spinlocks bereit, realisiert durch die Variable *pthread_mutex_t* und Funktionen mit Präfix *pthread_spin_*.

Ein Spinlock kann ebenfalls durch atomare Operationen der Hardware umgesetzt werden. In [Hof09] wurde gezeigt, dass sich die Laufzeit irregulärer taskbasierter Applikationen verbessert, wenn Spinlocks auf Basis atomarer Operationen verwendet werden.

In [KR07b] wurden ursprünglich Mutex-Locks aus der Pthread-Bibliothek eingesetzt. Die aktuell im Kandidatenpool vorhandenen spezialisierten Implementierungen wurden insofern verbessert, dass diese anstatt der Mutexe die Spinlocks auf Basis atomarer Operationen nutzen. Die Abbildung 9.4 zeigt unterschiedliche Implementierungen eines Lockmechanismus im Vergleich.

Unter Voraussetzung, dass jedem Thread ein ihm fest zugeordneter CPU-Kern zur Verfügung steht, der während des (aktiven) Wartens auf einen Lock nicht für andere Berechnungen verwendet wird, erzielt man deutlich bessere Laufzeiten, wenn anstelle eines Mutexes ein Spinlock verwendet wird. Der Spinlock auf Basis atomarer Operationen und der Pthread-Spinlock liefern ähnliche Laufzeiten.

Bei spezialisierten Implementierungsvarianten speichert jeder Thread lokal seine n/p Komponenten plus $2d(\mathbf{f})$ Komponenten, die er von seinen Nachbarn kopiert. Da die Funktion $\mathbf{f}$ eine beschränkte Zugriffsdistanz besitzt, können die m Matrizen $Y^{(k)} = \left(\mathbf{Y}_1^{(k)}, \ldots, \mathbf{Y}_s^{(k)}\right)^T \in \mathbb{R}^{s,n}$ mit $k = 1, \ldots, m$ zu einer Matrix $\bar{Y} = (\bar{y}_{l,j})$ der Größe $s \times$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
$(\boldsymbol{\eta}_\kappa, \ldots, \boldsymbol{\eta}_\kappa) = Y^{(0)}$																																
$Y^{(1)}$	1	2	4	7	11	16	21	26	26	21	16	11	7	4	2	1	1	2	4	7	11	16	21	26	26	21	16	11	7	4	2	1
$Y^{(2)}$	3	5	8	12	17	22	27	31	31	27	22	17	12	8	5	3	3	5	8	12	17	22	27	31	31	27	22	17	12	8	5	3
$Y^{(3)}$	6	9	13	18	23	28	32	35	35	32	28	23	18	13	9	6	6	9	13	18	23	28	32	35	35	32	28	23	18	13	9	6
$\Delta\boldsymbol{\eta}^{(m-1)}, Y^{(4)}$	10	14	19	24	29	33	36	38	38	36	33	29	24	19	14	10	10	14	19	24	29	33	36	38	38	36	33	29	24	19	14	10
$\Delta\boldsymbol{\eta}^{(m)}$	15	20	25	30	34	37	39	40	40	39	37	34	30	25	20	15	15	20	25	30	34	37	39	40	40	39	37	34	30	25	20	15
	P_1								P_2								P_3								P_4							

Abbildung 9.5.: Die Arbeitsweise paralleler Pipelining-Varianten. Die Kästchen in der Abbildung entsprechen den Blöcken der Matrizen $Y^{(k)}$ und den Vektoren $\Delta\boldsymbol{\eta}^{(m)}$ und $\Delta\boldsymbol{\eta}^{(m-1)}$. Die Nummerierung gibt die Reihenfolge der Berechnungen wieder. Die fett umrandeten Blöcke werden von den Nachbarthreads benötigt.

$(m-1) \cdot (B + d(\mathbf{f})) + 2d(\mathbf{f}) + n$ überlappt werden. Dabei ist B die Blockgröße, die das ODE-System in $n_B = n/B$ Blöcke mit $1 \leq B \leq n/p$ unterteilt.

Besitzt ein ODE-Problem eine beschränkte Zugriffsdistanz, so kann der Arbeitsraum der äußersten Schleife verkleinert werden, indem die Korrektorschritte gemäß dem Pipelining-Ansatz berechnet werden. Dazu wird die Schleifenstruktur der Implementierung *PipeDb1m* so verändert, dass die Schleife j zur äußersten Schleife wird. Wird das ODE-System in $n_B = n/B$ Blöcke unterteilt und gilt $B \geq d(\mathbf{f})$, so reichen maximal drei Blöcke des vorherigen Korrektorschrittes $k-1$, um einen Block des Korrektorschrittes k zu berechnen. Um eine effiziente Kommunikation gewährleisten zu können, bewegen sich benachbarte Prozessoren bei der Berechnung der Pipeline-Blöcke auf einander zu, d. h. sie arbeiten seitenverkehrt. Die Prozessoren mit gerader Nummer berechnen zuerst die Blöcke mit höherem Index und erst dann die Blöcke mit niedrigem Index. Die Prozessoren mit ungerader Nummer arbeiten in umgekehrter Reihenfolge. Der Austausch von $d(\mathbf{f})$ Komponenten ist nur für die Blöcke der Initialisierungs- bzw. Finalisierungsphase notwendig, die an den Grenzen zu Nachbarprozessoren liegen. Die Abbildung 9.5 veranschaulicht die Arbeitsweise der parallelen Pipelining-Varianten *ppDb1m*, *ppDb1mt*, *ppDb1myl* und *ppDb1mtyl*. Das Verwenden von Mutex-Variablen stellt sicher, dass die Berechnung eines Blocks abgeschlossen ist, bevor der Nachbarprozess auf diesen lesend zugreifen darf.

Die in [KR07b] präsentierten parallelen Pipelining-Varianten *ppDb1m*, *ppDb1mt* nutzen eine blockweise Datenverteilung auf der Basis von Blöcken der Größe $B \geq d(\mathbf{f})$. In allen Phasen der Pipeline wird die gleiche Blockgröße benutzt. Das allgemeine Pipeline-Berechnungsschema erlaubt jedoch die Verwendung unterschiedlicher Blockgrößen für die einzelnen Phasen der Pipeline. Während der Initialisierungs- bzw. Finalisierungsphase muss die Blockgröße der Bedingung $B \geq d(\mathbf{f})$ genügen. Ansonsten würden die Nachbarthreads Blöcke unterschiedlicher Größe berechnen und deshalb mit unterschiedlicher Geschwindigkeit arbeiten. Dadurch kann es vorkommen, dass ein Block der Größe $d(\mathbf{f})$ vom Nachbarthread gelesen werden muss, obwohl er noch nicht ausgerechnet wurde und es deshalb zu Geschwindigkeitseinbußen kommen kann. Für die Wahl der Blockgröße in der Sweep-Phase gilt diese Einschränkung nicht mehr. In dieser Phase kann auch

$B < d(\mathbf{f})$ als Blockgröße gewählt werden. Die Ergebnisse der Funktionsauswertungen des Blocks J im Korrektorschritt k müssen dann nicht um $2B$, sondern um $B + d(\mathbf{f})$ versetzt in der Matrix $\bar{Y}$ gespeichert werden (vgl. Abb. 7.6).

Die im Kandidatenpool enthaltenen Pipelining-Varianten wurden im Rahmen dieser Dissertation geändert. Die Varianten nutzen für die Initialisierungs- und Finalisierungs-Phase die kleinstmögliche Blockgröße $B_{i/f} = d(\mathbf{f})$, damit die Anzahl der in der Sweep-Phase berechneten Komponenten maximiert wird. Die Blockgröße B_s bleibt ein freier Implementierungsparameter, der zur Erzielung einer hohen Performance sorgfältig gewählt werden muss. Die Wahl der Blockgröße B_s für die Pipeline-Implementierungsvarianten ist eine Teil des Autotuning-Ansatzes.

Zusätzlich zu den vorgenommenen Verbesserungen bei den Implementierungsvarianten *ppDb1m* und *ppDb1mt* sind neue Implementierungsvarianten mit Suffix **yl* erstellt worden. Alle davor präsentierten Implementierungsvarianten für den gemeinsamen Adressraum, die aus [KR07b] stammen, nutzen gemeinsame Datenstrukturen, die von einem Thread allokiert werden und von allen anderen verwendet werden. Neue Varianten mit dem Suffix **yl* nutzen eine verteilte Abspeicherung des Argumentvektors $\mathbf{y}$.

Als nächstes wird der selbstadaptive parallele ODE-Solver für gemeinsamen Adressraum beschrieben.

9.3. Ein selbstadaptiver paralleler ODE-Solver für gemeinsamen Adressraum

Im Vergleich zum sequentiellen Autotuning, das in Kapiteln 7 und 8 präsentiert wurde, ist die automatische Auswahl von parallelen Varianten und von für diese geeigneten Optimierungsparametern mit weiteren Herausforderungen verbunden:

- Durch verschiedene Möglichkeiten zur Realisierung eines parallelen Programms steigt die Anzahl von Implementierungskandidaten stark an. Es kann eine Vielzahl paralleler Implementierungsvarianten generiert werden, die sich in den verwendeten Datenstrukturen, der gewählten Parallelisierungsstrategie (vgl. Abschnitt 3.5), der Datenverteilung und nicht zuletzt den Kommunikations- bzw. Synchronisationsoperationen unterscheiden.

- Die Unterschiede in den Laufzeiten einzelner Implementierungsvarianten sind größer als dies bei sequentiellen Varianten der Fall ist. Beispielsweise ist es auf einem großen System mit Hunderten von CPU-Kerne und gemeinsamem Speicher nicht ungewöhnlich, dass die beste Implementierungsvariante zehnmal so schnell läuft wie die langsamste Variante (s. Abschnitt 9.5.2 und 9.5.3). Werden sehr langsame Varianten zur Laufzeit evaluiert, so hat dies einen großen Einfluss auf die insgesamt resultierende Effizienz und Skalierbarkeit des Autotuners.

- Multicore-Systeme besitzen eine komplexe Speicherhierarchie. Eine oder mehrere Cache-Stufen können unter den Kernen eines Prozessors geteilt werden.

- Welche Implementierungsvariante die schnellste ist, kann zusätzlich von der Anzahl gestarteter Threads abhängen.
- Die Laufzeit paralleler Programme wird zusätzlich durch die verwendeten Synchronisationsoperationen bestimmt.

Im Fokus der Entwicklung eines selbstadaptiven ODE-Solvers für die parallelen Varianten mit gemeinsamem Adressraum steht die Auswahl effizienter Implementierungsvarianten und geeigneter Blockgrößen. Die Möglichkeit der Optimierung auf die Anzahl der parallel rechnenden Threads wurde nicht implementiert (anders als beim Autotuner für parallele Varianten mit verteiltem Adressraum, s. Kapitel 11). Bei der Implementierung des Solvers ist die Annahme getroffen worden, dass die Anzahl parallel rechnender Threads vom Benutzer fest vorgegeben ist und im Laufe des Programms nicht verändert wird. Dies hat mehrere Gründe:

- Wird der selbstadaptive Solver auf dem eigenen Arbeitsplatzrechner gestartet oder einem Server, den man alleine nutzen kann, so kann die Threadanzahl zur Laufzeit problemlos verändert werden. Dagegen kann auf großen Hochleistungsrechnern, die von vielen Nutzern gemeinsam verwendet werden, die Threadanzahl nach dem Programmstart nicht ohne Weiteres geändert werden. Die Benutzer solcher Systeme dürfen die Jobs nicht direkt starten, sondern übergeben diese an das sog. Batchsystem. Vor der Übergabe eines Jobs muss der Anwender in der Job-Beschreibung u. a. explizit angeben, wie viele Kerne er für die Berechnung benötigt und für wie lange. Sobald der Job an das Batchsystem abgeschickt worden ist, kann die Job-Beschreibung nicht mehr nachträglich geändert werden. Alle Jobs werden zunächst in eine Warteschlange gestellt. Oft stehen mehrere Warteschlangen zur Abspeicherung von Jobs zur Verfügung. Das Batchsystem verwaltet die Systemressourcen und arbeitet die Jobs in den Warteschlangen ab. Der Job-Scheduler des Batchsystems übernimmt die Zuteilung der Systemressourcen (Speicher, CPU-Kerne) an einzelne Anwender.
- Ein weiterer Grund ist, dass auf NUMA-Systemen eine Änderung der Threadanzahl zur Laufzeit teuer ist, da dadurch die Daten zwischen den Threads umverteilt werden müssen. Damit auf einem NUMA-System die Anzahl der Zugriffe auf entfernte Speichermodule gering ausfällt, müssen die Daten zwischen den Threads umverteilt werden. Die von einem Thread genutzten Daten müssen von anderen Threads in seinen lokalen Speicher kopiert werden.

9.3.1. Änderungen im Vergleich zum Autotuner für sequentielle Varianten

Zur Auswahl effizienter paralleler Varianten mit gemeinsamem Adressraum und für diese geeigneten Blockgrößen wurde das sequentielle Autotuning-Verfahren aus Kapitel 8 in folgenden Punkten erweitert:

- Der Algorithmus wurde um die Phase des Offline-Profilings ergänzt, die zur Installationszeit stattfindet. In dieser Phase wird anhand kleiner Benchmarks die

Laufzeit von Synchronisations-Operationen (Barriers) und vom Testproblem unabhängigen Berechnungen gemessen.

- In der Vorauswahlphase wird nicht nur geprüft, welche parallele Varianten aus dem Kandidatenpool auf das zu lösende Testproblem anwendbar sind, sondern auch welche Varianten mit den Eingabeparametern des Benutzers, speziell der Größe des ODE-Systems n und der Threadanzahl p, gestartet werden können. Zum Beispiel sind die spezialisierten Implementierungsvarianten mit Präfix $PipeDb1*$ nur dann einsetzbar, wenn die Anzahl der ODE-Gleichungen pro Thread größer oder gleich der Zugriffsdistanz ist, d. h. $n/p \geq d(\mathbf{f})$ gilt.
- Eine weitere Erweiterung stellt die Phase des Online-Profilings dar. Diese wird zum Messen der Ausführungszeit solcher Codeabschnitte genutzt, deren Laufzeit von der Eingabe abhängt. Beispiel dafür sind zusätzliche Operationen der Zeitschrittkontrolle, die von einigen Implementierungsvarianten vorgenommen werden.
- Die Blockgrößenauswahlstrategie wurde verändert. Der in Abschnitt 8.4.2 präsentierte modellbasierte Ansatz zur Ermittlung guter Blockgrößen für sequentielle Varianten entsprach im Wesentlichen einer Minimumstrategie, bei der der größte Arbeitsraum in die schnellstmögliche Cache-Stufe platziert werden soll, in die er noch hineinpasst. Diese lieferte gute Blockgrößenvorschläge, die nahe an das Optimum heranreichen (s. Kapitel 8.5.1). Ein Nachteil dieser Strategie ist jedoch, dass die gleiche Blockgröße auch für andere Arbeitsräume verwendet wird, für die u. U. eine größere Blockgröße noch eine Verbesserung der Laufzeit bewirkt hätte. Generell ist es so, dass die Minimumstrategie die Optimierung der Blockgröße bezüglich des L1-Caches favorisiert und deshalb dazu neigt, zu kleine Blockgrößen zurückzuliefern. Aus diesem Grund wurde der Ansatz der Blockgrößenauswahl für die parallelen Varianten dahingehend verbessert, dass die Blockgrößen mit Hilfe eines Modells für alle Arbeitsräume i und alle Cache-Stufen j vorselektiert werden. Zur Laufzeit werden dann nicht nur zwei Blockgrößen ts_{min} und ts_{small} empirisch evaluiert, sondern die Blockgrößen ts_{ij} in aufsteigender Reihenfolge.
- Es wurde eine Laufzeitvorhersage zum Ausschluss langsamer Varianten aus dem Kandidatenpool implementiert.

Nachfolgend wird die algorithmische Struktur des parallelen selbstadaptiven ODE-Solvers für gemeinsamen Adressraum präsentiert.

9.3.2. Die Zeitschritt-Phasen des ODE-Solvers

Die zeitschrittorientierte Berechnungsstruktur des selbstadaptiven parallelen ODE-Solvers für gemeinsamen Adressraum beinhaltet 5 Phasen:

- Die **Offline-Profiling-Phase** findet zur Installationszeit statt. Eine Reihe kleiner Benchmarks wird auf der Zielplattform ausgeführt und deren Ausführungszeit gemessen. Die gewonnenen Laufzeitinformationen werden dazu genutzt, die Dauer

der vom Testproblem unabhängigen Berechnungen und Synchronisationsoperationen vorab abzuschätzen.

- Die **Vorauswahlphase** geschieht vor dem Ausführen des ersten Zeitschrittes und dient dazu, alle die Implementierungsvarianten aus dem anfänglichen Kandidatenpool C_{all} zu entfernen, die entweder auf das zu lösende ODE-Problem nicht anwendbar sind oder mit den beim Start übergebenen Programmparametern des Benutzers (Größe des ODE-Systems, Threadanzahl) nicht gestartet werden können.

- Die **Online-Profiling-Phase** läuft nach der Vorauswahlphase und ebenfalls vor dem Ausführen des ersten Zeitschrittes ab. In dieser wird anhand kleiner Benchmarks die Ausführungszeit solcher Codeabschnitte gemessen, deren Laufzeit von der Eingabe abhängt.

- Die **Autotuningphase** wird während der ersten Zeitschritte des Integrationsprozesses durchgeführt. Die Anzahl der Autotuning-Schritte hängt von der Menge der zu evaluierten Implementierungskandidaten und Blockgrößen ab. In dieser Phase werden nacheinander Implementierungsvarianten aus dem Kandidatenpool C_P zur Berechnung einzelner Zeitschritte $\kappa = 1, 2 \ldots,$ herangezogen. Eine Implementierungsvariante $c \in C_P$ ohne Loop-Tiling wird zur Berechnung eines Zeitschrittes genutzt. Dagegen wird eine Loop-Tiling-Variante zur Berechnung mehrerer Zeitschritte verwendet, um für diese eine gute Blockgröße entsprechend der speziell entwickelten Blockgrößenauswahlstrategie (s. Abschnitt 9.4) zu bestimmen. Für jeden ausgeführten Autotuning-Zeitschritt (außer dem Aufwärmschritt) wird die Laufzeit erfasst. Die beste Implementierungsvariante wird am Ende der Autotuningphase anhand des Vergleichs der aufgezeichneten Laufzeiten ermittelt.

- **Berechnungsphase.** In der Berechnungsphase werden die restlichen Zeitschritte mit der schnellsten Implementierungsvariante und der besten Blockgröße ausgeführt, die in der Autotuningphase ermittelt wurden.

Nachstehend folgt eine detaillierte Beschreibung der einzelnen Zeitschritt-Phasen. Den Pseudocode des selbstadaptiven parallelen ODE-Solvers für gemeinsamen Adressraum zeigt die Abbildung 9.6. Details der Implementierung der Hilfsfunktion *eval_impl()* sind im Abschnitt 8.4.2 gegeben.

9.3.3. Offline-Profiling-Phase

In der Offline-Profiling-Phase, die während der Installation des selbstadaptiven Solvers auf der Zielplattform abläuft, wird mit Hilfe kleiner Benchmarks die Zeit zum Ausführen einer Barrier-Operation für eine unterschiedliche Anzahl der Threads gemessen. In der Autotuningphase werden diese Messwerte als Stützpunkte für die lineare Interpolation genutzt, um so den entsprechenden Synchronisationaufwand einer Implementierungsvariante für die vom Benutzer gestartete Anzahl der Threads abschätzen zu können.

```
1:  Sei P die zu lösende Probleminstanz;
2:  Sei C_all die Kandidatenmenge von Implementierungsvarianten;
3:  Sei T_bar(p) die Zeit für eine Barrier-Operation mit der an der Programmberechnung beteiligten Anzahl der Threads p;

    // Hilfsfunktion zur Evaluierung einer
    // Menge von Implementierungsvarian-
    // ten

4:  evaluate_class(C, var c_best, var B_best, var T_best)
5:  {
6:      while (C ≠ ∅ ∧ t < t_e)
7:      {
8:          Wähle eine (beliebige) Implementierung c ∈ C;
9:          C ← C \ c;
10:         (B^c_best, T^c_best) ← eval_impl(c);
11:         if (T^c_best < T_best) (c_best, B_best, T_best) ← (c, B^c_best, T^c_best);
12:     }
13: }

    // 1. Vorauswahlphase

    // Wähle die Implementierungsvarian-
    // ten aus der Menge C_all aus, die auf
    //  P anwendbar sind und mit Eingabe-
    // parametern des Benutzers gestartet
    // werden können. Ordne diese anhand
    // der Anzahl verwendeter Barrier-
    // Operationen pro Korrektorschritt
    // verschiedenen Klassen zu.

14: C_P ← {c ∈ C_all | c is applicable to P};
15: (C^0_P, C^1_P, C^2_P) ← classify_num_bar(C_P);

    // 2. Online-Profiling-Phase

    // Messe die Laufzeit solcher Codeab-
    // schnitte, die nicht von der Funktion
    // compute_time_step() erfasst werden.

16: do_runtime_measurements();

    // 3. Autotuningphase

    // Initialize t and h

17: t ← t_0;
18: h ← h_init;

    // Aufwärmschritt

19: Wähle eine (beliebige) Impl. c ∈ C_P ohne Loop-Tiling aus;
20: B ← n;    // Benötigt keine Blockgröße,
    // nutze n als Platzhalter.
21: compute_time_step(c, B, t);
22: Führe Schrittweitenkontrolle aus;
23: Aktualisiere t, wenn der lokale Fehler klein genug ist;

    // Bestimmte die schnellste Implemen-
    // tierung und eine geeignete Blockgrö-
    // ße für diese.

24: T_best ← ∞; // Laufzeit der bis jetzt
    // schnellsten Implementierungsvarian-
    // te
25: evaluate_class(C^0_p, c_best, B_best, T_best);

26: if (T_best < m · T_bar(p)) goto hpc-phase;
27: evaluate_class(C^1_p, c_best, B_best, T_best);

28: if (T_best < m·2T_bar(p)) goto hpc-phase;
29: evaluate_class(C^2_p, c_best, B_best, T_best);

    // 4. Berechnungsphase

30: label hpc-phase;
31: while (t < t_e) // Verbleibende Teil des
    // Integrationsintervalls
32: {
33:     compute_time_step(c_best, B_best, t);
34:     Führe Schrittweitenkontrolle aus;
35:     Aktualisiere t, wenn der lokale Fehler klein genug ist;
36: }
```

Abbildung 9.6.: Die Zeitschritt-Struktur des selbstadaptiven ODE-Solvers.

Bei der Implementierung des selbstadaptiven Solvers wird davon ausgegangen, dass einem Thread ein CPU-Kern zugeordnet ist, d. h. die Threads sind an die CPU-Kerne gebunden. Zur Bestimmung einer möglichst kleinen Anzahl von Stützpunkten für die lineare Interpolation wird betrachtet, wie viele Threads untereinander eine Cache-Stufe teilen; dieser Wert wird als p_s bezeichnet. Beispielsweise ist für einen Deca-Core-Prozessor mit privaten L1- und L2-Cache sowie einem L3-Cache, der von allen 10 Kernen geteilt wird, $p_s = 10$. Um die Skalierbarkeit einer Barrier-Operation auf einer Multi-Core-CPU genau ermitteln zu können, werden alle Threadzahlen p, die kleiner oder gleich p_s sind, in die Menge der Stützpunkte aufgenommen, d. h. alle Threadzahlen $p = 1, \dots, p_s$. Um die Ausführungszeit einer Barrier-Operation für eine größere Anzahl von Threads, die auf mehreren Multi-Core-CPUs laufen, abschätzen zu können, wird die Zahl der als Stützpunkte verwendeten Threads ausgehend von p_s verdoppelt ($p = 2p_s, 4p_s, 8p_s, \dots,$), bis die Gesamtzahl der verfügbaren Kerne der Zielplattform erreicht ist. Ist die Gesamtzahl der Kerne eines Multi-Core-Systems kein Vielfaches von $p_s \cdot 2^i$, so wird diese Anzahl zusätzlich in die Menge der Stützpunkte aufgenommen.

9.3.4. Vorauswahlphase

Die Vorauswahlphase wird zur Laufzeit ausgeführt, aber noch vor dem ersten Integrationszeitschritt. Zu diesem Zeitpunkt ist das zu lösende AWP bereits bekannt. Das Ziel der Vorauswahl ist, nur diejenigen Varianten im Kandidatenpool zu belassen, die auf das gegebene AWP angewendet und mit den Laufzeitparametern des Anwenders gestartet werden können.

Die Nutzung von spezialisierten Implementierungsvarianten im Kandidatenpool C_{all} setzt das Vorliegen einer beschränkten Zugriffsdistanz voraus. Daher können spezialisierte Varianten nur dann in den problem-spezifischen Kandidatenpool C_P aufgenommen werden, wenn das AWP eine beschränkte Zugriffsdistanz aufweist. Allgemeine Implementierungsvarianten können zur Lösung beliebiger AWPs eingesetzt werden und werden daher immer in den problem-spezifischen Kandidatenpool übernommen. Bevor eine Variante in den problem-spezifischen Kandidatenpool aufgenommen wird, wird geprüft, ob diese mit den Eingabeparametern des Benutzers, insbesondere der Anzahl der Threads p und der Problemgröße n, ausgeführt werden kann. Zum Beispiel sind die spezialisierten Varianten mit Präfix *PipeDb1** nur dann einsetzbar, wenn die Anzahl der Gleichungen pro Thread größer oder gleich der Zugriffsdistanz ist, d. h. $n/p \geq d(\mathbf{f})$ gilt. Die Pipeline-Varianten (Präfix *pp**) können nur dann verwendet werden, wenn jedem Thread ausreichend viele Gleichungen zur Verfügung stehen, um eine komplette Pipeline aufbauen zu können, oder als Bedingung formuliert, $n/p \geq (m + 2) \cdot d(\mathbf{f})$ gilt.

9.3.5. Online-Profiling-Phase

Nach der Vorauswahlphase folgt die Online-Profiling-Phase, die ebenfalls vor dem ersten Integrationszeitschritt abläuft. Diese wird gebraucht, um die Laufzeit der Code-Abschnitte zu erfassen, die durch die Messungen der Autotuningphase nicht abgedeckt werden (s. Abschnitt 9.3.6). Während die Zeitmessung in der Autotuningphase nur die

durch die Gleichungen (3.26)–(3.27) gegebene verschachtelte Schleife umfasst, führen einige Implementierungsvarianten, wie z. B. die mit verteilter Abspeicherung des Approximationsvektors $\mathbf{y}$, einen zusätzlichen Code in der Schrittweitenkontrolle aus. Die Implementierungsvarianten mit verteilter Abspeicherung des Vektors $\mathbf{y}$ müssen im Fall, dass ein Zeitschritt in der Schrittweitenkontrolle akzeptiert wird, $d(\mathbf{f})$ Vektorkomponenten vom linken und rechten Nachbarn kopieren. In der Online-Profiling-Phase wird die erforderliche Laufzeit für diese Kopiervorgänge gemessen. Diese Information wird später in der Autotuningphase zur Bestimmung der schnellsten Variante verwendet. Die Online-Profiling-Phase wird nur dann ausgeführt, wenn mindestens eine Variante, die zusätzliche Messungen erfordert, nach der Vorauswahlphase im Kandidatenpool C_P enthalten ist.

9.3.6. Autotuningphase

Nach der Online-Profiling-Phase wird die Autotuningphase gestartet, in der die Performance von Implementierungsvarianten aus dem Kandidatenpool C_P und der vom Modell bestimmten Blockgrößen (s. Abschnitt 9.4) bewertet wird. Die Zeitschritte der Autotuningphase tragen zum Lösungsprozess bei. Die unterschiedlichen Implementierungen der geschachtelten Schleife mit den Gleichungen (3.26)–(3.27) sind als Programmfunktionen in der Programmiersprache C realisiert worden. Der selbstadaptive Solver fungiert als Treiber, der diese Funktionen bereitstellt und aufruft. Die Schrittweitenkontrolle wurde ebenfalls als ein Teil des Treibercodes implementiert. Der Loop-Tiling-Code ist parametrisiert, die Blockgrößen innerhalb der Loop-Tiling-Schleifen sind durch symbolische Konstanten repräsentiert. Dadurch kann der gleiche Code mit verschiedenen Blockgrößen ausgeführt werden und braucht nicht neu generiert zu werden, so dass die Evaluierung unterschiedlicher Blockgrößen zur Laufzeit möglich ist.

In der Autotuningphase werden Implementierungsvarianten aus dem Kandidatenpool nacheinander evaluiert, um die schnellste Implementierungsvariante mit einer guten Blockgröße für das gegebene AWP, die Anzahl der Threads, die Größe des ODE-Systems und die aktuelle Zielplattform zu finden.

Eine Variante aus dem Kandidatenpool C_P wird wie folgt evaluiert:

- Enthält diese kein Loop-Tiling-Code, so wird sie zum Berechnen genau eines Zeitschrittes verwendet.
- Ist sie eine Loop-Tiling-Variante, so wird die Blockgrößenauswahlstrategie aus Abschnitt 9.4 dazu eingesetzt, für diese Variante eine effiziente Blockgröße zu finden. Diese bestimmt zunächst mit Hilfe eines analytischen Modells eine kleine Auswahl potenziell effizienter Blockgrößen, aus der dann die beste Blockgröße anhand einer empirisch-heuristischen Suche ermittelt wird.

Der Vergleich verschiedener Implementierungsvarianten und Blockgrößen erfolgt wie schon beim sequentiellen Solver anhand der gemessenen Laufzeit pro Zeitschritt (s. Abschnitt 7.6). Der allererste Zeitschritt wird zum Aufwärmen der Cache-Hierarchie genutzt, indem eine beliebige Implementierungsvariante ohne Loop-Tiling ausgeführt wird,

ohne dass ihre Laufzeit erfasst wird. Es ist mindestens ein Aufwärmschritt notwendig, damit die Daten in die Cache-Hierarchie geladen werden und die Messung der Laufzeit einzelner Implementierungsvarianten von einem reproduzierbaren Anfangszustand der Cache-Hierarchie aus erfolgt (s. Abschnitt 7.6).

Der Autotuning-Overhead hängt nicht nur von der Anzahl der getesteten Konfigurationen ab, d. h. von Varianten und Blockgrößen, die zur Laufzeit evaluiert werden, sondern auch vom Zeitverlust, der durch die Auswertung von Varianten, die langsamer als die Beste sind, entsteht.

Wie bereits dargelegt, sind die größten Herausforderungen für die Realisierung eines Autotuning-Ansatzes für parallele Varianten: die große Anzahl von Implementierungskandidaten (aufgrund verschiedener Möglichkeiten zur Realisierung eines parallelen Programms) und die großen Unterschiede in ihrem Skalierbarkeitsverhalten.

Das Testen sehr langsamer Implementierungsvarianten zur Laufzeit würde die Effizienz des Autotuners sehr stark verringern. Um dies zu vermeiden, enthält der Autotuning-Ansatz eine Strategie, mit der langsame Varianten aus dem Kandidatenpool anhand einer Laufzeitabschätzung vorab aussortiert werden können.

Der Kandidatenpool besteht aus allgemeinen und spezialisierten Implementierungen. Spezialisierte Varianten benötigen zur Synchronisation nur Locks, während allgemeine Implementierungsvarianten ein oder zwei Barrier-Operationen pro Korrektorschritt erfordern. Daher können spezialisierte Varianten, je nach Größe des ODE-Systems und der Funktionsauswertungskosten, oft im Vergleich zu allgemeinen Implementierungsvarianten eine deutlich höhere Skalierbarkeit erreichen. Es würde sich lohnen, wenn man allgemeine Implementierungen mit schlechter Performance vorab erkennen kann, ohne dass diese auf der Zielplattform ausgeführt werden müssen. Die Evaluierung langsamer allgemeiner Varianten kann durch die Abschätzung des benötigten Synchronisationsaufwands vermieden werden. Dazu werden in der Offline-Profiling-Phase kleine Benchmarks ausgeführt. Diese liefern Informationen über die Laufzeit von Barrier-Operationen für eine unterschiedliche Anzahl der Threads (s. Abschnitt 9.3.3). Hat der Autotuner bereits die Laufzeit der schnellsten spezialisierten Implementierungsvariante mit einer geeigneten Blockgröße ermittelt, so müssen allgemeine Implementierungsvarianten, bei denen bereits der zu erwartende Synchronisationsaufwand größer ist als die Laufzeit der schnellsten Variante, nicht mehr evaluiert werden.

Es sei $T_{bar}(p)$ die Ausführungszeit einer Barrier-Operation mit p Threads und m die Anzahl der Korrektorschritte des verwendeten PC-Verfahrens. Die Autototuningphase läuft dann dann folgendermaßen ab:

- Alle Implementierungsvarianten aus dem Kandidatenpool werden entsprechend der Anzahl von Barrier-Operationen pro Korrektorschritt sortiert und dann in dieser Reihenfolge evaluiert.

- Als Erstes werden alle spezialisierten Varianten getestet, da diese keine Barrier-Operationen innerhalb eines Korrektorschrittes erfordern und deshalb i. d. R. gut skalieren. Nach diesem Schritt steht die Laufzeit der bis zu diesem Zeitpunkt schnellsten spezialisierten Implementierung fest.

- Dann werden allgemeine Varianten mit einer Barrier-Operation pro Korrektorschritt (Variante D und alle von ihr abgeleiteten Varianten) evaluiert, jedoch nur dann, wenn ihr abgeschätzter Synchronisationsaufwand $m \cdot T_{bar}(p)$ nicht größer als die Laufzeit der besten spezialisierten Implementierung ist. Ist eine allgemeine Variante mit einer Barrier-Operation pro Korrektorschritt schneller als die beste spezialisierte Variante, so wird T_{best} auf die Laufzeit dieser Variante gesetzt.

- Als Letztes werden allgemeine Implementierungsvarianten mit zwei Barrier-Operationen pro Korrektorschritt (A, E und *EAblock*) evaluiert, aber nur dann, wenn ihr vorausberechneter Synchronisationsaufwand $m \cdot 2T_{bar}(p)$ nicht größer als die Laufzeit der schnellsten Implementierung T_{best} ist. Der Wert T_{best} wird immer dann aktualisiert, wenn eine Implementierungsvariante eine geringere Laufzeit erzielt als alle davor ausgewerteten.

9.3.7. Berechnungsphase

Nachdem die schnellste Implementierungsvariante c_{best} mit einer effizienten Blockgröße B_{best} bestimmt worden ist, wird diese zur Berechnung verbliebener Zeitschritte genutzt.

9.4. Die Strategie zur Blockgrößenauswahl

In diesem Abschnitt wird die Blockgrößenauswahlstrategie beschrieben. Die Auswahl geeigneter Blockgrößen findet in der Autotuningphase statt.

9.4.1. Einfluss der Blockgröße auf die Performance paralleler Implementierungsvarianten mit Loop-Tiling

Viele Implementierungsvarianten aus dem Kandidatenpool nutzen die Optimierungstechnik des Loop-Tilings. Die Performance eines Loop-Tiling-Codes kann jedoch stark von der gewählten Blockgröße abhängen. Wird diese zu klein gewählt, so kann der Cache möglicherweise nicht optimal ausgenutzt werden und der zusätzliche Overhead der Schleifenkontrolle kann den Vorteil des Loop-Tilings untergraben. Wird die Blockgröße zu groß gewählt, so überschreitet die Größe des Arbeitsraums der dazugehörenden Schleife die Größe des Caches, was eventuell zu weniger Datenwiederverwendung und einer erhöhten Anzahl der Konfliktfehlzugriffe führen kann.

Die Abbildungen 9.7–9.10 zeigen für zwei Testprobleme, BRUSS2D und STRING, den Einfluss der Blockgröße auf die Performance von parallelen Implementierungsvarianten für einen gemeinsamen Adressraum. Die Experimente wurden auf den Testrechnern SGI UV und AMD Opteron 6172 (als „Leo "bezeichnet) mit den Loop-Tiling-Varianten *PipeDb2mt*, *PipeDb1mt* und *ppDb1mt* durchgeführt. Die Blockgröße wurde – ausgegangen von der Blockgröße 16 – so lange verdoppelt, bis die Anzahl der Komponenten pro Thread n/p erreicht wurde. Die Bilder zeigen, um wie viel Prozent eine bestimmte Blockgröße langsamer ist als die optimale Blockgröße in diesem Bereich. Der maximale Laufzeitver-

lust, der durch die Wahl einer langsamen Blockgröße entstehen kann, liegt zwischen 30% und 800%.

Diese Ergebnisse demonstrieren, dass bei parallelen Varianten die Wahl einer ungünstigen Blockgröße einen viel größeren Einfluss auf die Laufzeit haben kann als dies bei sequentiellen Varianten der Fall ist. In den Abbildungen ist deutlich zu erkennen, dass insbesondere große Blockgrößen nah an n die Laufzeit von Implementierungsvarianten sehr stark verschlechtern, aber auch die Wahl zu kleiner Blockgrößen kann in vielen Fällen die Performance paralleler Implementierungsvarianten verringern.

9.4.2. Blockgrößenauswahl

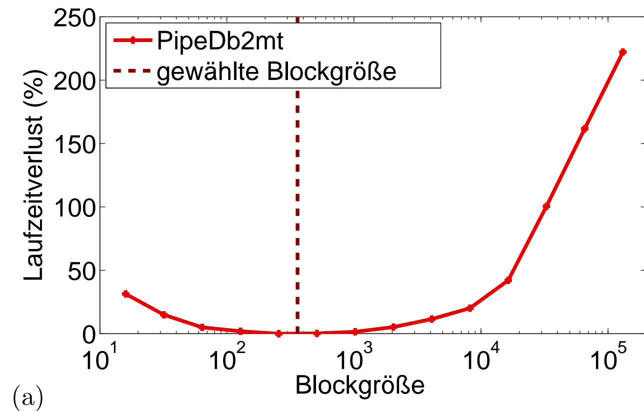
Im Kapitel 8 wurde bereits ein Ansatz zur automatischen Auswahl von Blockgrößen für sequentielle Varianten präsentiert. Dieser nutzt ein analytisches Modell zur Vorauswahl potenziell effizienter Blockgrößen aus, die anschließend zur Laufzeit auf dem Zielsystem evaluiert werden. Das Modell basiert auf der Bestimmung von Arbeitsräumen der in den Implementierungsvarianten auftretenden Schleifen und berücksichtigt bei der Wahl einer Blockgröße die relevanten Eigenschaften der Speicherhierarchie der Zielplattform. Das Blockgrößenauswahlmodell für sequentielle Varianten schlägt maximal zwei Blockgrößen für jede Implementierungsvariante vor, die dann in der Autotuningphase evaluiert werden.

Im parallelen Szenario sind die wichtigsten Herausforderungen bei der Realisierung eines Ansatzes zur automatischen Blockgrößenauswahl: die im Vergleich zur sequentiellen Ausführung höhere Anzahl von Implementierungskandidaten und die komplexe Cache-Hierarchie der Multi-Core-CPUs. Daher ist der Blockgrößenauswahl-Ansatz für sequentielle Varianten für die parallelen Implementierungsvarianten erweitert und modifiziert worden.

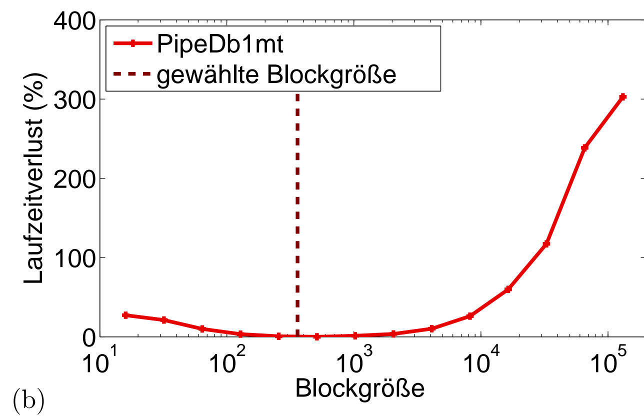
Zur Reduktion des Suchraumes potenzieller Blockgrößen wird erneut ein auf der Berechnung von Arbeitsräumen beruhendes analytisches Modell verwendet. Dieses wird aber derart erweitert, dass Blockgrößen für alle Arbeitsräume und Cache-Stufen vorselektiert werden. Die Menge vorausgewählter Blockgrößen enthält mehr als zwei Blockgrößen und die beste Blockgröße daraus wird anhand einer heuristisch-empirischen Suche bestimmt. Des Weiteren wird das Blockgrößenmodell an die Charakteristik der Cache-Hierarchie von Multi-Core-CPUs angepasst. Bei der Berechnung der Größe von Arbeitsräumen wird berücksichtigt, dass einige Caches von mehreren CPU-Kernen gemeinsam genutzt werden.

Das Arbeitsraum-Modell zur Vorauswahl von Blockgrößen

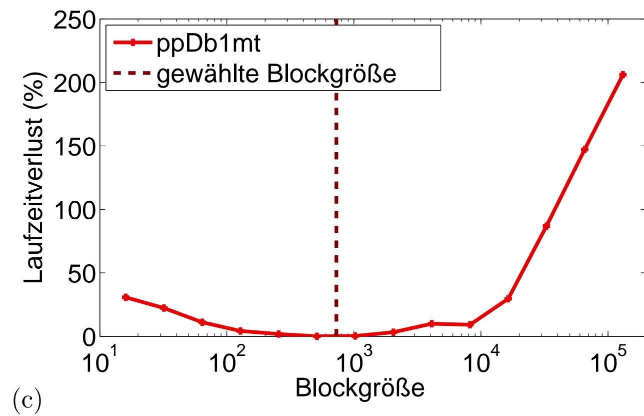
Um den Suchraum möglicher Blockgrößen einzuschränken, wird, wie schon bei sequentiellen Varianten, ein analytisches Modell verwendet. Dieses wählt für jede Variante eine Reihe von Blockgrößen aus, basierend auf der Abschätzung der Größe laufzeitrelevanter Arbeitsräume und Parametern der Cache-Hierarchie. Die wesentlichen Änderungen zum Arbeitsraum-Modell für sequentielle Implementierungsvarianten bestehen in Folgendem:



(a)



(b)



(c)

Abbildung 9.7.: Einfluss der Blockgröße auf die Performance von Implementierungsvarianten sowie die Qualität ausgewählter Blockgrößen. Testproblem BRUSS2D mit $n = 8.0 \cdot 10^6$ auf dem AMD Opteron 6172 System und 48 Threads. (a) *PipeDb2mt*, (b) *PipeDb1mt*, (c) *ppDb1mt*.

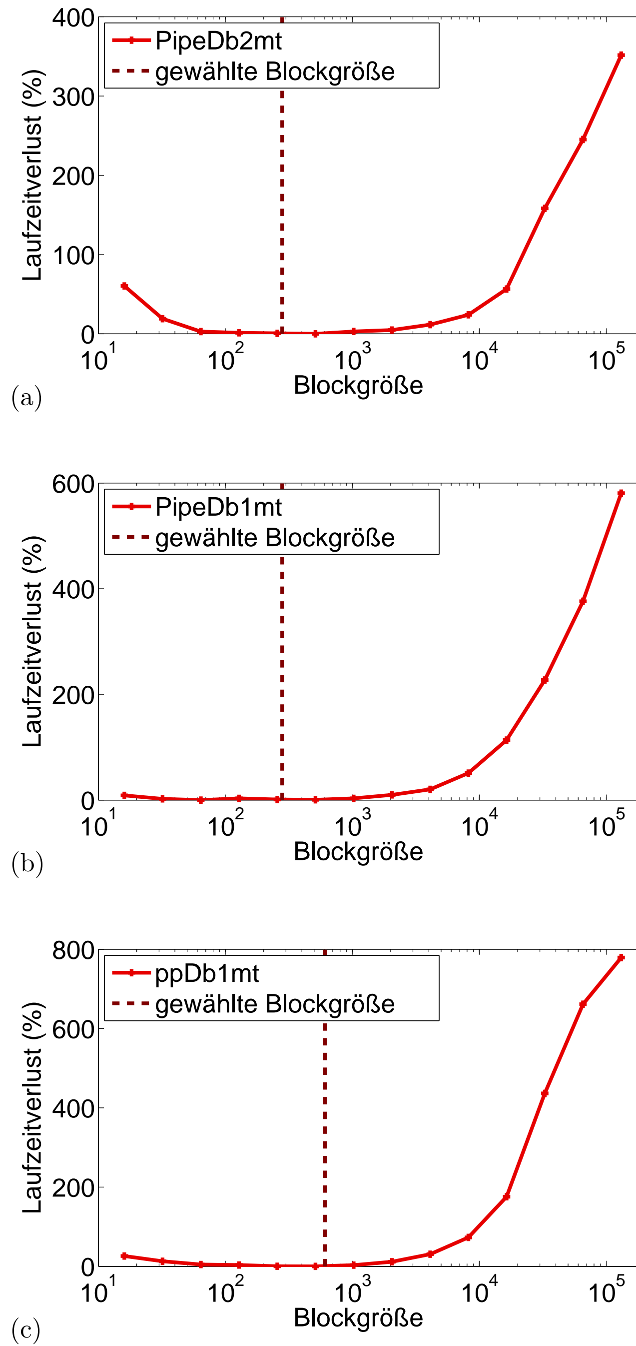


Abbildung 9.8.: Einfluss der Blockgröße auf die Performance von Implementierungsvarianten sowie die Qualität ausgewählter Blockgrößen. Testproblem STRING mit $n = 8.0 \cdot 10^6$ auf dem AMD Opteron 6172 System und 48 Threads. (a) *PipeDb2mt*, (b) *PipeDb1mt*, (c) *ppDb1mt*.

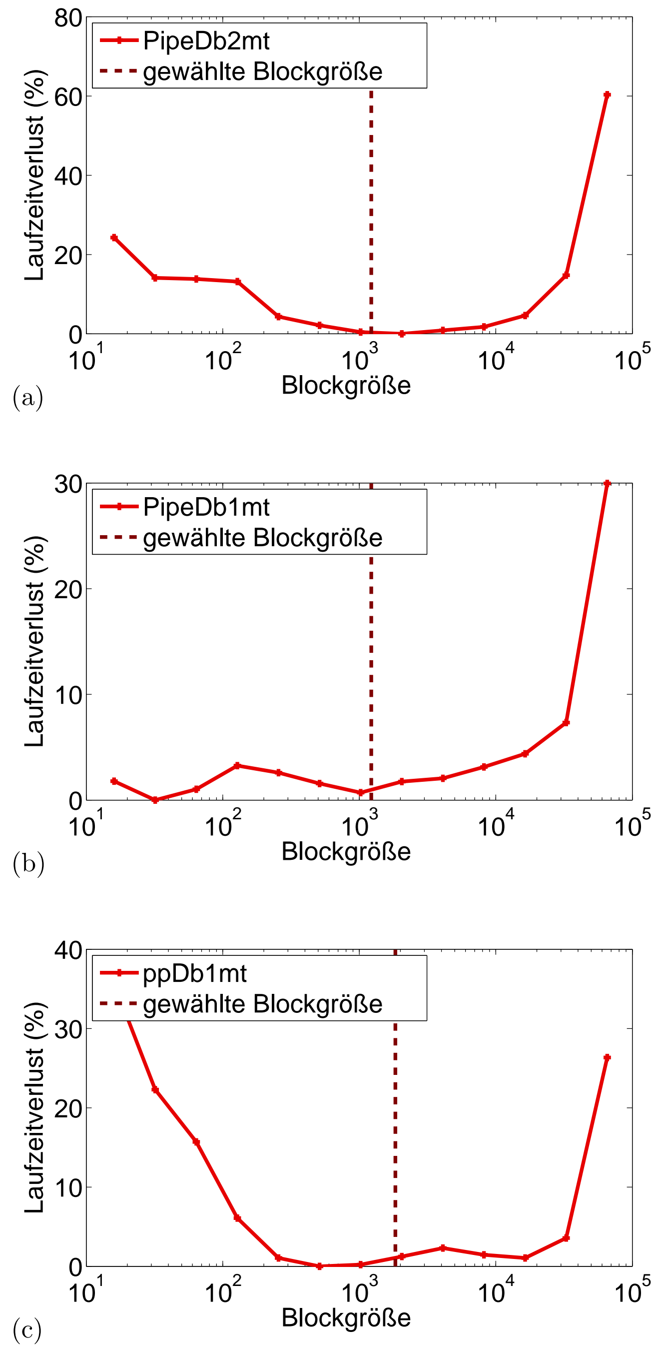


Abbildung 9.9.: Einfluss der Blockgröße auf die Performance von Implementierungsvarianten sowie die Qualität ausgewählter Blockgrößen. Testproblem BRUSS2D mit $n = 8.0 \cdot 10^6$ auf dem SGI UV System und 80 Threads. (a) *PipeDb2mt*, (b) *PipeDb1mt*, (c) *ppDb1mt*.

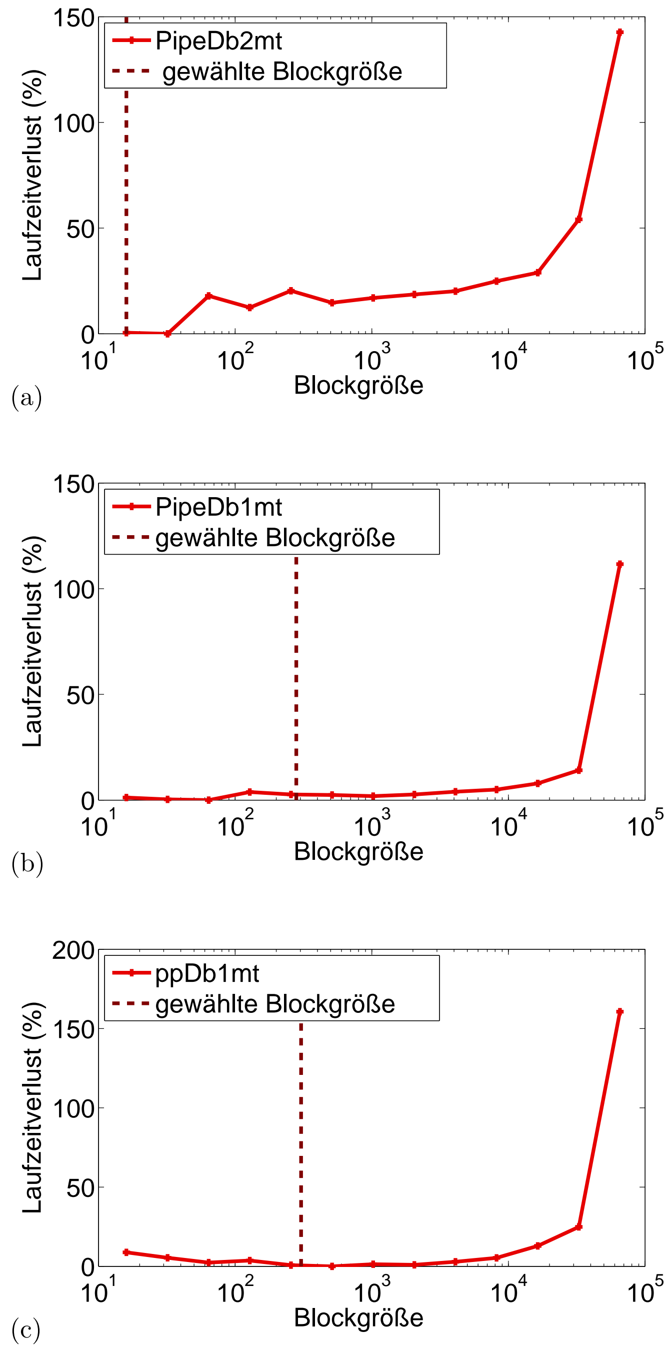


Abbildung 9.10.: Einfluss der Blockgröße auf die Performance von Implementierungsvarianten sowie die Qualität ausgewählter Blockgrößen. Testproblem STRING mit $n = 8.0 \cdot 10^6$ auf dem SGI UV System und 80 Threads. (a) *PipeDb2mt*, (b) *PipeDb1mt*, (c) *ppDb1mt*.

- Im Gegensatz zu sequentiellen Implementierungsvarianten muss man bei der Berechnung von Arbeitsräumen paralleler Varianten beachten, dass durch die blockweise Aufteilung jeder Thread nur einen Teil der Gleichungen des zu lösenden ODE-Systems berechnet. Dadurch ändert sich die Größe der Arbeitsräume von Schleifen, die über die Systemdimension iterieren. Innerhalb der Systemdimensionsschleifen greift jeder Thread p auf die ihm zugeteilten $\lceil n/p \rceil$ Komponenten zu und, falls Funktionsauswertungen innerhalb dieser Schleifen vorhanden sind, noch zusätzlich auf Vektorkomponente der anderen Threads. Allgemeine Varianten benötigen für die Funktionsauswertung eines Blocks von $\lceil n/p \rceil$ Komponenten alle Komponenten des Argumentvektors $\mathbf{y}$, während spezialisierte Varianten, die durch eine beschränkte Zugriffsdistanz $d(\mathbf{f}) \ll n$ charakterisiert sind, nur $\lceil n/p \rceil$ Komponenten plus jeweils $d(\mathbf{f})$ Komponenten brauchen, die vom rechten und linken Nachbarthreads kopiert werden müssen (siehe Abb. 9.3). Bei der Beschreibung von Arbeitsräumen wird die Konstante $D = 2d(\mathbf{f}) + 1$ verwendet. Diese gibt die maximale Anzahl benötigter Vektorkomponenten zur Funktionsauswertung einer Komponente f_j an. Die Gesamtzahl der während der Auswertung eines Blocks von $\lceil n/p \rceil$ Komponenten zugegriffenen Komponenten des Argumentvektors wird in den Arbeitsräumen durch $\lceil n/p \rceil + D - 1$ angegeben.

- Bei vielen modernen Multicore-Prozessoren wird die höchste Cache-Stufe von mehreren CPU-Kernen gemeinsam genutzt. Das Arbeitsmodell trifft die Annahme, dass die Kapazität eines gemeinsamen Caches gleichmäßig unter den Threads aufgeteilt ist.

  Besitzt ein ODE-System eine beschränkte Zugriffsdistanz und wird zur Lösung eine allgemeine Variante verwendet, so wird bei der Funktionsauswertung von $\lceil n/p \rceil$ Komponenten eines Threads implizit auf $d(\mathbf{f})$ weitere Komponenten jedes seiner Nachbarthreads zugegriffen. Diese zusätzlichen Komponenten sind ein Teil des Arbeitsraumes eines Zeit- bzw. Korrektorschrittes und benötigen im Fall getrennter Caches ein Teil der Kapazität zum Ablegen von $2 \cdot d(\mathbf{f})$ Komponenten. Wird ein Cache jedoch von benachbarten Threads geteilt, so sind $2 \cdot d(\mathbf{f})$ Komponenten bereits als ein Teil der $\lceil n/p \rceil$ Komponenten der beiden Threads im gemeinsam genutzten Cache vorhanden.

  Das Blockgrößenauswahlmodell verwendet einen zusätzlichen Parameter, um sich die Anzahl der Threads zu merken, die eine Cache-Stufe teilen. Die Anzahl der Threads, die ein Cache der Stufe L_i teilen, wird s_i bezeichnet. Steht eine Cache-Stufe L_i exklusive nur einem CPU-Kern zur Verfügung, so wird $s_i = 1$ gesetzt.

- Es werden Blockgrößen für alle Arbeitsräume und Cache-Stufen vorselektiert.

Die Tabellen 9.3 und 9.4 zeigen die laufzeitrelevanten Arbeitsräume, die für einzelne Loop-Tiling-Varianten ausgewählt wurden. Dabei bezeichnet B die Blockgröße, n/p ist die Anzahl der Komponenten pro Thread, s ist die Anzahl der Stufen und m die Anzahl der Korrektorschritte des verwendeten ODE-Verfahrens, $d(\mathbf{f})$ ist die Zugriffsdistanz und s_i die Anzahl der Threads, die sich ein Cache der Stufe L_i teilen.

ID	Loop	Size
Dblock		
WS_{ts}	Zeitschritt	$[(2s+3)\lceil n/p \rceil + B] \cdot s_i + (2s+1) \cdot 2d$
WS_{cs}	Korrektorschritt	$[(2s+1)\lceil n/p \rceil + B] \cdot s_i + (2s+1) \cdot 2d$
WS_1	Eine Iter. der i Schleife	$[(s+2)\lceil n/p \rceil + B] \cdot s_i + (s+2) \cdot 2d$
WS_2	Eine Iter. der j-Schleife $(k = 1, \ldots, m-2)$	$[(s+3)B + 2d] \cdot s_i$
WS_3	Eine Iter. der l-Schleife $[(jj)]$	$2B \cdot s_i$
WS_4	Eine Iter. der l-Schleife $[jj]$	$3B \cdot s_i$
PipeDb2mt		
WS_{ts}	Zeitschritt	$[(2s+3)\lceil n/p \rceil + B] \cdot s_i + (2s+1) \cdot 2d$
WS_{cs}	Korrektorschritt	$[(2s+1)\lceil n/p \rceil + B] \cdot s_i + (2s+1) \cdot 2d$
WS_1	Eine Iter. der j-Schleife $(k = 1, \ldots, m-2)$	$[(2s+3)B + s \cdot 2d] \cdot s_i$
WS_2	Eine Iter. der i-Schleife	$[(s+3)B + 2d] \cdot s_i$
WS_3	Eine Iter. der l-Schleife $[(jj)]$	$2B \cdot s_i$
WS_4	Eine Iter. der l-Schleife $[jj]$	$3B \cdot s_i$
PipeDb2m		
WS_1	Eine Iter. der j Schleife $(k = 1, \ldots, m-2)$	$[(2s+3)B + s \cdot 2d] \cdot s_i$
WS_2	Eine Iter. der i-Schleife	$[(s+2)B + 2d] \cdot s_i$

Tabelle 9.3.: Ausgewählte Arbeitsräume für parallele allgemeine Loop-Tiling-Varianten; dabei ist $d = d(\mathbf{f})$.

ID	Loop	Size
PipeDb1m		
WS_{ts}	Zeitschritt	$[s \cdot ((m-1)(B+d) + \lceil n/p \rceil) + 3\lceil n/p \rceil + s \cdot 2d] \cdot s_i + 2d$
WS_{cs}	Korrektorschritt	$[(s+1) \cdot \lceil n/p \rceil + s \cdot (B+3d)] \cdot s_i + 2d$
WS_1	Eine Iter. der j-Schleife ($k = 1, \ldots, m-2$)	$[(2s+2) \cdot B + s \cdot 2d] \cdot s_i$
WS_2	Eine Iter. der i-Schleife	$[(s+2) \cdot B + 2d] \cdot s_i$
PipeDb1mt		
WS_{ts}	Zeitschritt	$[s \cdot ((m-1)(B+d) + \lceil n/p \rceil) + 3\lceil n/p \rceil + s \cdot 2d + B] \cdot s_i + 2d$
WS_{cs}	Korrektorschritt	$[(s+1) \cdot \lceil n/p \rceil + s \cdot (B+3d) + B] \cdot s_i + 2d$
WS_1	Eine Iter. der j-Schleife ($k = 1, \ldots, m-2$)	$[(2s+3) \cdot B + s \cdot 2d] \cdot s_i$
WS_2	Eine Iter. der i-Schleife	$[(s+3) \cdot B + 2d] \cdot s_i$
WS_3	Eine Iter. der l-Schleife $[(jj)]$	$2B \cdot s_i$
WS_4	Eine Iter. der l-Schleife $[jj]$	$3B \cdot s_i$
ppDb1m		
WS_{ts}	Zeitschritt	$[s \cdot ((m-1)(b+d) + \lceil n/p \rceil) + 3\lceil n/p \rceil + s \cdot 2b + b] \cdot s_i + 2d$
WS_{ps}	Pipelining-Schritt	$[ms(B+2d) + (m+2)B + 3d] \cdot s_i$
WS_1	Eine Iter. der j-Schleife ($k = 1, \ldots, m-2$)	$[(2s+2)B + s \cdot 2d] \cdot s_i$
WS_2	Eine Iter. der j-Schleife ($k = m-1$)	$[(2s+1)B + s \cdot 2d] \cdot s_i$
WS_3	Eine Iter. der i-Schleife	$[(s+2)B + 2d] \cdot s_i$
ppDb1mt		
WS_{ts}	Zeitschritt	$[s \cdot ((m-1)(b+d) + \lceil n/p \rceil) + 3\lceil n/p \rceil + s \cdot 2b + b] \cdot s_i + 2d$
WS_{ps}	Pipelining-Schritt	$[ms(B+2d) + (m+2)B + 3d] \cdot s_i$
WS_1	Eine Iter. der j-Schleife ($k = 1, \ldots, m-2$)	$[(2s+3)B + s \cdot 2d] \cdot s_i$
WS_2	Eine Iter. der j-Schleife ($k = m-1$)	$[(2s+2)B + s \cdot 2d] \cdot s_i$
WS_3	Eine Iter. der i-Schleife	$[(s+3)B + 2d] \cdot s_i$
WS_4	Eine Iter. der l-Schleife $[(jj)]$	$2B \cdot s_i$
WS_5	Eine Iter. der l-Schleife $[jj]$	$3B \cdot s_i$

Tabelle 9.4.: Ausgewählte Arbeitsräume für parallele spezialisierte Loop-Tiling-Varianten; dabei ist $d = d(\mathbf{f})$ und $b = \max(B, d(\mathbf{f}))$.

Die Vorauswahl einer Menge von Blockgrößen mit dem erweiterten Arbeitsraum-Modell läuft wie folgt ab:

- Bestimme für jede Loop-Tiling-Variante die Menge laufzeitrelevanter Arbeitsräume $\mathcal{W}$.

- Für jeden Arbeitsraum $w \in \mathcal{W}$ und jede Cache-Stufe $i \in \{1, \ldots, r\}$ berechne die Funktion $\mathrm{LT}(w, C_{\mathrm{eff}}(\mathrm{L}i), s_i)$. Diese gibt die größte Blockgröße an, für die der Arbeitsraum w in die Cache-Stufe i passt. Überschreitet die Größe des Arbeitsraumes w die (u. U. geteilte) Kapazität der Cache-Stufe i, so wird $\mathrm{LT}(w, C_{\mathrm{eff}}(\mathrm{L}i), s_i)$ gleich der Anzahl der Komponenten pro Thread gesetzt, n/p:

$$\mathrm{LT}(w, C_{\mathrm{eff}}(\mathrm{L}i), s_i) = \begin{cases} \max\{B \in \{1, \ldots, \min(n/p, C_{\mathrm{eff}}(\mathrm{L}i),)\} \mid \\ \qquad \text{ws_size}(w, B, s_i) \leq C_{\mathrm{eff}}(\mathrm{L}i)\}, \\ \qquad \text{wenn } \exists\, B : \text{ws_size}(w, B, s_i) \leq C_{\mathrm{eff}}(\mathrm{L}i), \\ n/p, \quad \text{sonst.} \end{cases} \tag{9.1}$$

Die Hilfsfunktion $\text{ws_size}(w, B, s_i)$ berechnet die Größe des kumulierten Arbeitsraumes w mit der Blockgröße B, der von s_i Threads gebildet wird, die untereinander die Cache-Stufe $\mathrm{L}i$ teilen:

$$\text{ws_size}(w, B, s_i) = \begin{cases} \text{Größe des Arbeitsraumes } w & \\ \text{mit der Blockgröße } B & \\ \text{und Threadzahl } s_i, & \text{wenn } B \in \{1, \ldots, n/p\}, \\ n/p, & \text{sonst.} \end{cases} \tag{9.2}$$

- Da die Daten im Cache in Form von Cachezeilen abgelegt werden, werden die durch die Funktion $\mathrm{LT}(w, C_{\mathrm{eff}}(\mathrm{L}i), s_i)$ berechneten Blockgrößen auf das nächst kleinere Vielfache einer Cachezeile l_i der betrachteten Cache-Stufe $\mathrm{L}i$ gerundet:

$$\mathrm{LT}_{l_i}(w, C_{\mathrm{eff}}(\mathrm{L}i), s_i) = \left\lfloor \frac{\mathrm{LT}(w, C_{\mathrm{eff}}(\mathrm{L}i), s_i)}{l_i} \right\rfloor \cdot l_i.$$

- Nehme alle im vorherigen Schritt berechneten Blockgrößen $\mathrm{LT}_{l_i}(w, C_{\mathrm{eff}}(\mathrm{L}i), s_i)$ in die Menge vorselektierter Blockgrößen $\mathcal{T}$ auf.

Die Abbildung 9.11 fasst die Methode der Vorauswahl von Blockgrößen zusammen.

Empirische Suche auf der Menge vorausgewählter Blockgrößen

Die Minimumstrategie, die zur Bestimmung geeigneter Blockgrößen für sequentielle Loop-Tiling-Varianten eingesetzt wurde (s. Abschnitt 8.4.2), versucht, den größten Arbeitsraum in die schnellstmögliche Cache-Stufe zu platzieren, in die er noch hinein passt.

1: pre-select_tile_sizes(c)
2: {
3: Sei $\mathcal{W} = \{\mathrm{WS}_{\mathrm{ts}}, \mathrm{WS}_{\mathrm{cs}}, \mathrm{WS}_1, \mathrm{WS}_2, \ldots\}$ die Menge laufzeitrelevanter Arbeitsräume einer Implemementierungsvariante c, die von der Blockgröße B abhängen.
4: Sei $\mathcal{C} = \{\mathrm{L1}, \mathrm{L2}, \ldots \mathrm{L}r\}$ die Menge der Cache-Stufen, l_i die Blocklänge der Cache-Stufe Li und s_i die Anzahl der Threads, die sich die Kapazität der Cache-Stufe Li teilen.
5: Sei $\mathcal{T}$ die zu bestimmende Menge vorausgewählter Blockgrößen.

6: **if** $(\mathcal{W} = \varnothing)$ *// c ist eine Variante ohne Loop-Tiling.*
7: **return** $\{n/p\}$;

8: $\mathcal{T} \leftarrow \varnothing$;
9: **for each** $(i \in \{1, \ldots, r\})$
10: **for each** $(w \in \mathcal{W})$
11: $\mathcal{T} \leftarrow \mathcal{T} \cup \mathrm{LT}_{l_i}(w, C_{\mathrm{eff}}(\mathrm{L}i), s_i)$;
12: **return** $\mathcal{T}$;
13: }

Abbildung 9.11.: Die Hilfsfunktion pre-select_tile_sizes() als Teil der Autotuningphase. Diese liefert anhand des Arbeitsraum-Modells zu jeder Implementierungsvariante c eine Menge vorselektierter Blockgrößen.

Somit bevorzugt sie die Blockgrößen, die für den L1-Cache generiert werden, und neigt aus diesem Grund zu kleinen Blockgrößen. Sie liefert gute Blockgrößenvorschläge, die nah an das Optimum heranreichen, verwendet aber die gleiche Blockgröße auch für andere Arbeitsräume, für die u. U. eine größere Blockgröße noch eine Verbesserung der Laufzeit bewirkt hätte. In einigen Laufzeitexperimenten wurde beobachtet, dass größere Blockgrößen für andere Cache-Stufen oder kleinere Arbeitsräume zu besseren Laufzeiten führen können. Zudem nutzt das Arbeitsmodell für sequentielle Varianten eine auf praktischer Erfahrung beruhende Heuristik für den Vorschlag eines festen Wertes für die zweite Blockgröße (s. Abb. 8.4, Zeile 6). Ein einzelner fester Wert reicht jedoch zum Abbilden unterschiedlicher Eigenschaften verschiedener Hardware-Plattformen und Testprobleme nicht aus. Daher wurde die Blockgrößenauswahlstrategie für die parallelen Loop-Tiling-Varianten um eine heuristisch-empirische Suche erweitert.

Nach Vorauswahl von Blockgrößen mit Hilfe des Arbeitsraummodells wird die beste Blockgröße aus dieser Menge anhand empirischer Suche ermittelt. Dazu werden die N Blockgrößen aus der Vorauswahlmenge T zunächst in aufsteigender Reihenfolge sortiert, so dass $\mathcal{T} = \langle B_1, B_2, \ldots, B_N \rangle$ mit $B_i < B_j$ für $1 \leq i < j \leq N$ gilt. Dann werden die Blockgrößen in dieser Reihenfolge solange evaluiert, bis die aktuell bewertete Blockgröße B_i langsamer ist als die davor evaluierte Blockgröße B_{i-1}. Die Blockgröße B_{i-1} ist dann die beste Blockgröße in der Menge T.

Auf einigen Prozessoren und für einige AWP-Probleme sind kleine Blockgrößen manchmal effizienter als eine Blockgröße, die einen großen Teil des Ziel-Speichers (für welchen

die Blockgröße gewählt wurde) verwendet. Kommt die empirische Suche zum Ergebnis, dass B_1 die schnellste Blockgröße in der Menge T ist, so wird die Suche mit der Evaluierung von Blockgrößen kleiner als B_1 fortgesetzt. Für die Bewertung der Blockgrößen kleiner als B_1 wird die Blockgröße in jedem Schritt halbiert und evaluiert, bis die Laufzeit der Implementierungsvariante mit der halbierten Blockgröße sich nicht mehr verbessert. Die Abbildung 9.12 zusammen mit der Abbildung 9.13 zeigt den Pseudocode des verwendeten modellgeführten Suchverfahrens.

```
 1: evaluate_implementation(c)
 2: {
 3:     𝒯 ← pre-select_tile_sizes(c);
 4:     B_min ← min 𝒯;
 5:     T^c_best ← ∞;

    // Evaluiere die Blockgrößen aus der Menge 𝒯 in aufsteigender Reihenfolge.

 6:     stop ← false;
 7:     while (𝒯 ≠ ∅ ∧ t < t_e ∧ ¬stop)
 8:     {
 9:         B ← min 𝒯;
10:         𝒯 ← 𝒯 \ B;
11:         stop ← eval_tile_size(c, B, B^c_best, T^c_best);
12:     }

    // Wenn die kleinste Blockgröße in 𝒯 bis jetzt die schnellste ist,
    // so betrachte noch kleinere Blockgrößen.

13:     if (B^c_best = B_min)
14:     {
15:         B ← ⌊(B_min/2) / l_min⌋ · l_min;
16:         stop ← false;
17:         while (B ≥ l_min ∧ t < t_e ∧ ¬stop)
18:         {
19:             stop ← eval_tile_size(c, B, B^c_best, T^c_best);
20:             B ← ⌊(B/2) / l_min⌋ · l_min;
21:         }
22:     }

23:     return (B^c_best, T^c_best);
24: }
```

Abbildung 9.12.: Hilfsfunktion evaluate_implementation() als Teil der Autotuningphase. Diese nutzt zur Bestimmung einer effizienten Blockgröße für eine Implementierungsvariante c eine modellgeführte Suche.

```
1: eval_tile_size(c, B, var B^c_best, var T^c_best)
2: {
3:     T ← runtime_of(compute_time_step(c, B, t));
4:     correct_runtime_using_profile_info(T);

5:     führe Schrittweitenkontrolle aus;
6:     aktualisiere t, wenn der lokale Fehler klein genug;

7:     if (T < T^c_best für die Mehrheit der Threads)
8:     {
9:         (B^c_best, T^c_best) ← (T, B);
10:        return false;
11:    }
12:    else
13:        return true;
14: }
```

Abbildung 9.13.: Hilfsfunktion eval_tile_size() als Teil der Autotuningphase. Diese misst die Laufzeit einer Implementierungsvariante c unter Verwendung der Blockgröße B.

Ermittlung der besten Blockgröße anhand der Mehrheitsentscheidung

In einem Multithread-Programm liefern die Threads leicht unterschiedliche Laufzeiten bei der Auswertung ein und der selben Blockgröße. Bei der Fragestellung, ob die aktuell ausgewertete Blockgröße schneller ist als die vorher evaluierte, können die einzelnen Threads auf Grund gemessener Laufzeiten zu unterschiedlichem Ergebnis kommen. In diesem Szenario stellt sich die Frage, wie man die Entscheidung trifft, ob die modellgeführte Suche fortgesetzt wird oder nicht.

Der Autotuning-Algorithmus lässt die Mehrzahl der Threads entscheiden, ob die Blockgröße aus dem aktuellen Zeitschritt schneller ist als die Vorherige und deshalb die Suche fortgeführt wird. Zu diesem Zweck wird ein gemeinsamer Array `vote` mit einem Eintrag für jeden Thread verwaltet, der die Entscheidung repräsentiert, ob die Suche fortgesetzt oder gestoppt wird. Wenn der Thread i dafür stimmt, dass die Suche mit der nächsten Blockgröße fortgesetzt wird, so wird `vote[i]` auf 1 gesetzt. Andernfalls wird `vote[i]` auf 0 gesetzt. Bevor die Threads mit der Evaluierung der nächsten Blockgröße fortfahren, zählt der Thread 0 alle Stimmen zusammen und stoppt die Suche, wenn mehr als 50 % der Threads mit `vote[i]` $= 0$ abgestimmt haben.

Im nächsten Abschnitt folgt eine experimentelle Evaluierung des entworfenen selbstadaptiven ODE-Solvers.

9.5. Experimentelle Evaluierung

In diesem Abschnitt wird eine experimentelle Evaluierung des selbstadaptiven parallelen ODE-Solvers für gemeinsamen Adressraum und der integrierten Blockgrößenauswahlstrategie präsentiert. Die Experimente wurden auf drei unterschiedlichen Rechnersystemen durchgeführt, deren Beschreibung man im Anhang B findet. Auf allen Testplattformen wurde der C++ Compiler aus der GNU-Compiler-Collection (GCC) benutzt, auf der SGI UV in der Version 4.6.1, auf Leo in der Version 4.7.1 und auf dem 2-Socket-Nehalem-System in der Version 4.7.3. Auf allen Testplattformen wurden die Optimierungsstufe `-O3` und die Compilerflags `-D_REENTRANT -fPIC -std=c++0x -D_STDC_LIMIT_MACROS -g -rdynamic` verwendet.

Als Testprobleme wurden BRUSS2D [HNW09a] und STRING [HNW09a] genutzt (s. Anhang).

9.5.1. Experimentelle Validierung der Blockgrößenauswahlstrategie

Detaillierte Untersuchung anhand ausgewählter Testfälle

Zur Validierung der Blockgrößenauswahlstrategie wurde die Ausführungszeit der Varianten *PipeDb2mt*, *PipeDb1mt* und *ppDb1mt* unter der Variation der Blockgröße auf verschiedenen Testrechnern und für unterschiedliche ODE-Probleme gemessen. Angefangen von der Blockgröße 16 (entspricht zwei Cachezeilen von jeweils 64 Byte) wurde die Blockgröße jeweils verdoppelt, bis die maximale Anzahl der Komponenten pro Thread (n/p) erreicht war.

Die Bilder 9.7 und 9.8 zeigen für unterschiedliche Blockgrößen den entstehenden Performance-Verlust bezogen auf die Laufzeit mit der optimalen Blockgröße (innerhalb des Bereichs abgetasteter Blockgrößenwerte). Die Experimente wurden auf dem Leo-System mit 48 Threads und der Problemgröße $n = 8.0 \cdot 10^6$ durchgeführt. Außerdem ist in den Abbildungen die durch die Blockgrößenauswahlstrategie bestimmte Blockgröße eingezeichnet, sodass bestimmt werden kann, wie nah diese an die optimale Blockgröße heranreicht.

Es zeigt sich, dass für die Implementierungsvariante *PipeDb2mt* und das BRUSS2D-Problem (Abb. 9.7 (a)) die Wahl sehr kleiner Blockgrößen ungünstig ist, weil es dadurch zu Performance-Einbußen von bis zu 30 % im Vergleich zu der optimalen Blockgröße kommen kann. Die Laufzeit der Implementierung mit der Blockgröße Zum Erreichen einer guten Performance muss die Blockgröße aus dem Bereich $[64, 2048]$ gewählt werden, denn ihre Laufzeit ist nur 5 % schlechter als die der optimalen Blockgröße. Ab der Blockgröße 2048 steigt die Laufzeit der Implementierung stets an, bedingt dadurch, dass laufzeitrelevante Arbeitsräume nach und nach aus bestimmten Cache-Stufen herausfallen und sich dadurch die Kosten der Speicherzugriffe verteuern.

Die Blockgrößenauswahlstrategie evaluiert fünf Blockgrößen $\langle 1456, 2456, 728, 360, 176 \rangle$ (in dieser Reihenfolge) und wählt am Ende 360 als die beste Blockgröße aus. Die Blockgrößen 1456 und 2456 sind Vorschläge des Arbeitsraummodells. Weil die kleinste Blockgröße 1456 aus der Menge vorausgewählter Blockgrößen $\mathcal{T}$ gleichzeitig auch die schnells-

te ist, werden auch Blockgrößen kleiner als 1456 evaluiert. Die am Ende ausgewählte Blockgröße 360 kommt sehr nah an die optimale Blockgröße 256 heran, die im Laufe der Validierung abgetastet wurde. Die nächstgrößere Blockgröße 512 würde zu einem Leistungsabfall von 0,2 % führen.

Ähnlich wie bei der Implementierung *PipeDb2mt* kann man auch bei den anderen beiden Varianten *PipeDb1mt* und *ppDb1mt* (Abb. 9.7 (b) und (c)) beobachten, dass kleine Blockgrößen nicht optimal sind und zu einem Leistungsabfall von bis zu 30 % gegenüber der besten abgetasteten Blockgröße führen können. Allerdings scheint die Wahl einer ungünstigen Blockgröße einen stärkeren Einfluss auf die Laufzeit von Implementierungen *PipeDb1mt* und *ppDb1mt* zu haben als auf die Laufzeit der Implementierung *PipeDb2mt*. Die Wahl einer Blockgröße größer als 16384 würde bei der Implementierung *PipeDb1mt* einen Leistungsabfall in Höhe von 60–320 % und bei der Implementierung *ppDb1mt* in Höhe von 30–206 % bewirken.

Für die Variante *PipeDb1mt* wird von der Auswahlstrategie ebenfalls die Blockgröße 360 gewählt. Die Performance dieser Blockgröße ist nur ca. 0.5 % schlechter als die der optimalen Blockgröße 512. Für die Variante *ppDb1mt* gibt die Strategie 728 als Blockgröße zurück, diese kommt ebenfalls sehr nah ($\approx$ 0.1 %) an die Performance der optimalen Blockgröße (512) heran.

Vergleicht man die BRUSS2D- und STRING-Probleme (Abb. 9.7 und 9.8) untereinander, so lässt sich bei der Nutzung nicht-optimaler, insbesondere großer Blockgrößen ein höherer Performance-Verlust für das STRING-Problem beobachten. Betrachtet man beispielsweise die Implementierung *ppDb1mt*, so beträgt der Leistungsabfall beim STRING-Problem maximal 778 % und beim BRUSS2D-Problem maximal 206 % gegenüber der optimalen Blockgröße. Dass beide Probleme unterschiedlich stark auf die Wahl einer ungünstigen Blockgröße reagieren, lässt sich vor allem durch die Unterschiede im Zugriffsmuster der Funktion der rechten Seite begründen. Beim BRUSS2D wird für die Funktionsauswertung einer Komponente auf fünf Komponenten des Argumentvektors zugegriffen, die durch den 5-Punkt-Stern gegeben und innerhalb der Zugriffsdistanz $d(\mathbf{f}) = 2N$ zu finden sind. Beim STRING greift die Funktionsauswertung einer Komponente der rechten Seite nur auf zwei Komponenten des Argumentvektors zu, die innerhalb der Zugriffsdistanz $d(\mathbf{f}) = 3$ liegen.

Die Qualität der durch die Blockgrößenauswahlstrategie bestimmten Blockgrößen ist bei beiden Probleme ähnlich hoch. Für die Implementierungsvarianten *PipeDb2mt* und *PipeDb1mt* (Abb. 9.8 (a) und (b)) wird die Blockgröße 280 gewählt, die im Fall der Implementierung *PipeDb2mt* $\approx$ 0.5 % und im Fall der Implementierung *PipeDb1mt* $\approx$ 1 % schlechter als die optimale Blockgröße ist. Auch für Implementierungsvariante *ppDb1mt* (Abb. 9.8 (c)) wird eine geeignete Blockgröße gewählt, deren Laufzeit nur $\approx$ 0.2 % von der optimalen Blockgröße entfernt ist.

Die Bilder 9.9 und 9.10 zeigen die gewählten Blockgrößen für drei Implementierungsvarianten *PipeDb2mt*, *PipeDb1mt* und *ppDb1mt* und zwei Testprobleme auf der zweiten Architektur SGI UV und unter Verwendung von 80 Threads. Für das BRUSS2D-Problem (Abb. 9.9) werden Blockgrößen gewählt, deren Laufzeit nur $\approx$ 0.4–1.7 % höher ist als die optimale Blockgröße der jeweiligen Variante. Auch beim STRING-Problem (Abb. 9.10)

ist die Performance ermittelter Blockgrößen nur $\approx 0.5 - 2.6\,\%$ vom Optimum entfernt.

Verifikation der Blockgrößenauswahlstrategie anhand weiterer Testfälle

Zusätzlich zur detaillierten Validierung der Blockgrößenauswahlstrategie der in den Abbildungen 9.7 und 9.10 präsentierten Experimente wurde die Effektivität der Blockgrößenauswahlstrategie in mehreren Laufzeitexperimenten auf verschiedenen Zielsystemen mit unterschiedlicher Architektur und mit unterschiedlicher Anzahl der Threads für verschiedene Testprobleme und Problemgrößen untersucht.

Die Tabellen 9.5 und 9.6 fassen die Ergebnisse dieser Testfälle zusammen. In den Tabellen sind die Anzahl ausgewerteter Blockgrößen, der Leistungsverlust der ausgewählten Blockgröße gegenüber der optimalen und der schlechtesten Blockgröße ausgewiesen.

Die Ergebnisse zeigen, dass die Performance der durch die Blockgrößenauswahl-strategie ermittelten Blockgrößen sehr nah an das Optimum heran reicht. Für alle außer einem getesteten Benchmark ist die Performance der ausgewählten Blockgrößen nur 0–7 % vom Optimum entfernt. Nur in einem Testfall (Implementierungsvariante *PipeDb2mt*, Testproblem STRING mit $n = 8.0 \cdot 10^6$, 160 Threads, SGI UV System) liegt die Performance 25 % vom Optimum entfernt, was aber nach wie vor akzeptabel ist im Vergleich zu dem Leistungsabfall von 66 %, der durch die Wahl der schlechtesten Blockgröße entstehen würde. Bei diesem Benchmark wird die beste Performance für sehr kleine Blockgrößen erreicht. Die beste Blockgröße ist 16, welche gleichzeitig auch die kleinste abgetastete Blockgröße ist. Die Laufzeit der nächst größeren Blockgröße 32 ist nur 6 % schlechter. Für noch größere Blockgrößen beträgt der Laufzeitverlust 18–66 %. Die Blockgrößenauswahlstrategie evaluiert die vom Arbeitsmodell vorgeschlagenen Blockgrößen $\langle 280, 456, 1224, 1840\rangle$ und liefert 1224 als die beste Blockgröße zurück. Weil die Blockgröße 1224 größer ist als die kleinste durch das Arbeitsmodell vorselektierte Blockgröße 280, werden die Blockgrößen kleiner als 280 nicht durchsucht und die Blockgrößenauswahlstrategie bleibt im lokalen Minimum gefangen.

In allen Testfällen ist die Anzahl der zur Laufzeit evaluierten Blockgrößen sehr klein. Für jede Implementierungsvariante werden zwischen 3 und 7 Blockgrößen getestet. Diese Anzahl reicht aber aus, um eine geeignete Blockgröße zu bestimmen, deren Performance im Durchschnitt 2–3 % und im schlechtesten Fall 25 % von der Optimalen entfernt ist.

Um zu verifizieren, dass die Blockgrößenauswahlstrategie auch für andere Korrektorverfahren erfolgreich funktioniert, sind auf dem 2-Socket-Nehalem-System Experimente mit Radau I A(5)-Verfahren durchgeführt worden. Die Tabelle 9.7 zeigt die entsprechenden Resultate. Auch für das 3-stufige Korrektorverfahren Radau I A(5) erzielt die Blockgrößenauswahlstrategie ähnlich gute Ergebnisse, wie für das 5-stufige Verfahren Lobatto III C(8).

Einfluss der Vektorisierung auf die Wahl geeigneter Blockgrößen

Die meisten modernen Prozessoren enthalten neben den herkömmlichen Integer- und Gleitkomma-Einheiten zusätzlich SIMD-Einheiten (s. Kapitel 6.3), welche eine gleichzeitige Ausführung von arithmetischen Operationen auf mehreren Datensätzen erlauben.

				Laufzeitverlust	
n	Implementierung	p	♯ Blockgrößen	Gewählte zur Besten	Schlechteste zur Besten
BRUSS2D auf Leo					
$0.5 \cdot 10^6$	*PipeDb1mt*	48	3	1 %	31 %
$0.5 \cdot 10^6$	*PipeDb2mt*	48	4	2 %	18 %
$0.5 \cdot 10^6$	*ppDb1mt*	48	3	4 %	12 %
$8.0 \cdot 10^6$	*PipeDb1mt*	24	5	1 %	350 %
$8.0 \cdot 10^6$	*PipeDb2mt*	24	5	2 %	273 %
$8.0 \cdot 10^6$	*ppDb1mt*	24	4	1 %	230 %
STRING auf Leo					
$0.5 \cdot 10^6$	*PipeDb1mt*	48	3	1 %	84 %
$0.5 \cdot 10^6$	*PipeDb2mt*	48	3	1 %	26 %
$0.5 \cdot 10^6$	*ppDb1mt*	48	5	1 %	79 %
$8.0 \cdot 10^6$	*PipeDb1mt*	24	4	1 %	658 %
$8.0 \cdot 10^6$	*PipeDb2mt*	24	5	1 %	426 %
$8.0 \cdot 10^6$	*ppDb1mt*	24	3	1 %	962 %
BRUSS2D auf SGI UV					
$0.5 \cdot 10^6$	*PipeDb1mt*	80	5	2 %	2 %
$0.5 \cdot 10^6$	*PipeDb2mt*	80	4	1 %	21 %
$0.5 \cdot 10^6$	*ppDb1mt*	80	—[a]	—[a]	—[a]
$8.0 \cdot 10^6$	*PipeDb1mt*	160	6	1 %	48 %
$8.0 \cdot 10^6$	*PipeDb2mt*	160	3	1 %	41 %
$8.0 \cdot 10^6$	*ppDb1mt*	160	3	7 %	47 %
$8.0 \cdot 10^6$	*PipeDb1mt*	240	6	4 %	248 %
$8.0 \cdot 10^6$	*PipeDb2mt*	240	4	2 %	21 %
$8.0 \cdot 10^6$	*ppDb1mt*	240	6	7 %	130 %
STRING auf SGI UV					
$0.5 \cdot 10^6$	*PipeDb1mt*	80	5	0 %	3 %
$0.5 \cdot 10^6$	*PipeDb2mt*	80	3	2 %	3 %
$0.5 \cdot 10^6$	*ppDb1mt*	80	4	1 %	8 %
$8.0 \cdot 10^6$	*PipeDb1mt*	160	5	1 %	107 %
$8.0 \cdot 10^6$	*PipeDb2mt*	160	4	25 %	66 %
$8.0 \cdot 10^6$	*ppDb1mt*	160	4	1 %	178 %
$8.0 \cdot 10^6$	*PipeDb1mt*	240	5	0 %	296 %
$8.0 \cdot 10^6$	*PipeDb2mt*	240	3	4 %	37 %
$8.0 \cdot 10^6$	*ppDb1mt*	240	4	5 %	405 %

[a] Nicht genug Komponente pro Thread zum Aufbau einer vollen Pipeline vorhanden.

Tabelle 9.5.: Validierung der Blockgrößenauswahlstrategie auf zwei Testplattformen unter Verwendung des Lobatto III C(8)-Verfahrens.

n	Implementierung	p	♯ Blockgrößen	Laufzeitverlust Gewählte zur Besten	Laufzeitverlust Schlechteste zur Besten
BRUSS2D auf 2-Socket-Nehalem					
$0.5 \cdot 10^6$	*PipeDb1mt*	8	3	1 %	24 %
$0.5 \cdot 10^6$	*PipeDb2mt*	8	3	1 %	30 %
$0.5 \cdot 10^6$	*ppDb1mt*	8	5	1 %	14 %
$8.0 \cdot 10^6$	*PipeDb1mt*	8	3	1 %	86 %
$8.0 \cdot 10^6$	*PipeDb2mt*	8	3	1 %	96 %
$8.0 \cdot 10^6$	*ppDb1mt*	8	4	1 %	80 %
STRING auf 2-Socket-Nehalem					
$0.5 \cdot 10^6$	*PipeDb1mt*	8	4	1 %	42 %
$0.5 \cdot 10^6$	*PipeDb2mt*	8	3	1 %	46 %
$0.5 \cdot 10^6$	*ppDb1mt*	8	6	1 %	42 %
$8.0 \cdot 10^6$	*PipeDb1mt*	8	4	1 %	154 %
$8.0 \cdot 10^6$	*PipeDb2mt*	8	4	2 %	163 %
$8.0 \cdot 10^6$	*ppDb1mt*	8	7	1 %	162 %

Tabelle 9.6.: Validierung der Blockgrößenauswahlstrategie auf dem 2-Socket-Nehalem-System unter Verwendung des Lobatto III C(8)-Verfahrens.

n	Implementierung	p	♯ Blockgrößen	Laufzeitverlust Gewählte zur Besten	Laufzeitverlust Schlechteste zur Besten
BRUSS2D auf 2-Socket-Nehalem					
$0.5 \cdot 10^6$	*PipeDb1mt*	8	3	1 %	17 %
$0.5 \cdot 10^6$	*PipeDb2mt*	8	3	1 %	19 %
$0.5 \cdot 10^6$	*ppDb1mt*	8	4	1 %	11 %
$8.0 \cdot 10^6$	*PipeDb1mt*	8	3	1 %	60 %
$8.0 \cdot 10^6$	*PipeDb2mt*	8	3	1 %	65 %
$8.0 \cdot 10^6$	*ppDb1mt*	8	3	1 %	61 %
STRING on 2-socket-Nehalem					
$0.5 \cdot 10^6$	*PipeDb1mt*	8	3	1 %	31 %
$0.5 \cdot 10^6$	*PipeDb2mt*	8	4	1 %	31 %
$0.5 \cdot 10^6$	*ppDb1mt*	8	3	1 %	34 %
$8.0 \cdot 10^6$	*PipeDb1mt*	8	3	1 %	109 %
$8.0 \cdot 10^6$	*PipeDb2mt*	8	3	2 %	118 %
$8.0 \cdot 10^6$	*ppDb1mt*	8	4	1 %	126 %

Tabelle 9.7.: Validierung der Blockgrößenauswahlstrategie auf dem 2-Socket-Nehalem-System unter Verwendung des Radau I A(5)-Verfahrens.

Alle Implementierungsvarianten, die in inneren Schleifen über die Systemdimension iterieren (u. a. *Dblock*, *PipeDb2mt*, *PipeDb1mt* und *ppDb1mt*) sind grundsätzlich dafür geeignet, vom Compiler automatisch vektorisiert zu werden. In den Experimenten hat sich aber herausgestellt, dass die auf den Zielplattformen vorinstallierten Versionen des *GCC*-Compilers (4.6.1, 4.7.1 und 4.7.3) nicht in der Lage waren, diese Varianten automatisch zu vektorisieren. Zudem sind alle drei ausgewählten Architekturen zwar mit modernen Prozessoren ausgestattet, stellen aber keine AVX-Befehlssatzerweiterungen (s. Kapitel 6.3) zu Verfügung.

Um den Effekt der Vektorisierung auf die Laufzeit von Implementierungsvarianten mit verschiedenen Blockgrößen zu untersuchen, wurden ergänzend Experimente auf einem Rechnersystem durchgeführt, das zwar nur 4 Kerne enthält, dafür aber mit einem Intel-Core i7-4770 Prozessor ausgestattet ist. Dieser besitzt eine Haswell-Mikroarchitektur und unterstützt den AVX2-Befehlssatz. Als Compiler wurde der Intel C++ Compiler in der Version 14.0.1 und die Flags `-O2 -xHost` verwendet. Weil manche Implementierungsvarianten überlappenden Datenstrukturen nutzen und es dadurch für den Compiler nicht ersichtlich ist, ob die zu vektorisierende Schleife Datenabhängigkeiten enthält, wird dem Compiler explizit durch die Eingabe des Pragmas `#pragma ivdep` mitgeteilt, dass die Schleife vektorisiert werden kann.

Einige Ergebnisse sind in Abb. 9.14 und Tabelle 9.8 dargestellt. Diese zeigen, dass, wenn die Blockgröße so gewählt wird, dass die laufzeitrelevanten Arbeitsräume im Cache abgelegt sind, die Laufzeit des vektorisierten Codes 16 % schneller ist als die des nicht-vektorisierten. Im Fall einer ungünstigen Wahl der Blockgröße steigt die Anzahl der Cache-Fehlzugriffe an und die Leistung des Programms lässt sich nicht mehr mit Hilfe der Vektorisierung steigern. Das heißt, wenn die Vektorisierung verwendet wird, so ist die Wahl einer effizienten Blockgröße besonders wichtig. Der Bereich mit guten Blockgrößen ist ist zwar in beiden Fällen, mit und ohne Vektorisierung, ähnlich, zeigt jedoch Unterschiede auf. Ist die Vektorisierung aktiviert, so kann die Blockgröße mit der geringsten Laufzeit eindeutig identifiziert werden. Ist die Vektorisierung deaktiviert, so ähnelt die Laufzeitkurve dem horizontal gespiegelten Buchstaben “L”, weil es eine Reihe von Blockgrößen gibt, die eine ähnlich gute Effizienz aufweisen.

Insgesamt zeigen die Ergebnisse, dass die Blockgrößenauswahlstrategie gute Blockgrößen liefert, unabhängig davon, ob die Vektorisierung verwendet wird oder nicht.

Zusammenfassung der Ergebnisse

Die Laufzeitexperimente haben die Annahme bestätigt, dass der geeignete Wert für eine Blockgröße von dem zu lösendem AWP, der Implementierung, der verwendeten Threadanzahl und den Eigenschaften der Zielplattform abhängt.

Obwohl die Aufgabe der automatischen Blockgrößenauswahl recht anspruchsvoll ist, ist der entwickelte Blockgrößenauswahlansatz in der Lage, effiziente Blockgrößen für unterschiedliche Implementierungsvarianten, zu lösende AWPs und verschiedene Zielplattformen zu bestimmen. Die automatische Auswahl der Blockgrößen verursacht nur einen geringen Autotuning-Overhead, da nur einige wenige Blockgrößen zur Laufzeit evaluiert werden müssen. Die Experimente demonstrieren, dass zur Auswahl einer geeigneten

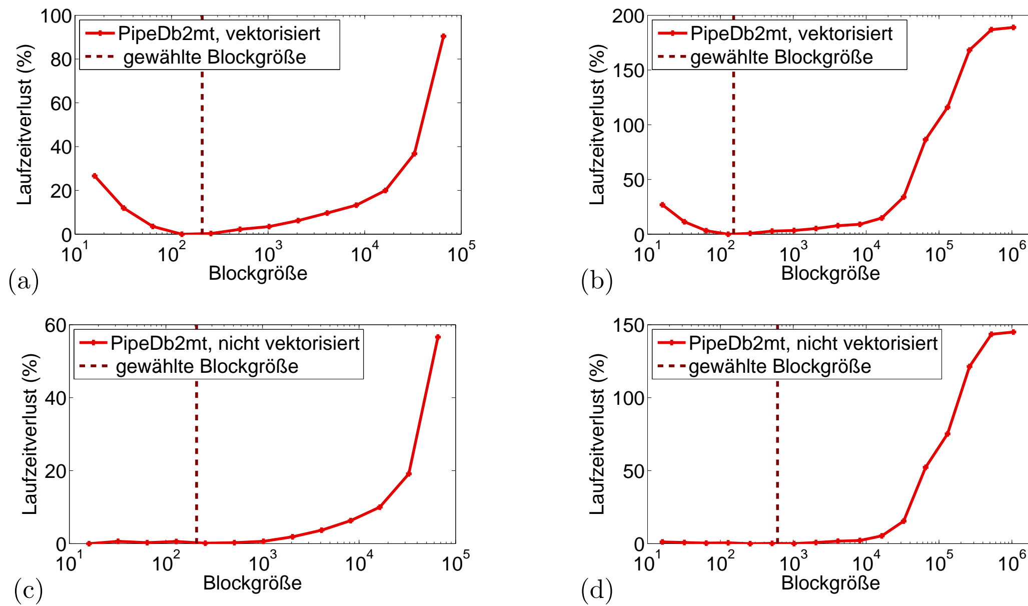


Abbildung 9.14.: Einfluss der Vektorisierung auf die Laufzeit von Blockgrößen und die gewählte Blockgröße. Testproblem: BRUSS2D. Korrektorverfahren: Lobatto III C(8). Testplattform: Intel Core i7-4770 (Haswell), 4 Threads. Compiler: icpc (ICC) 14.0.1. (a) $n = 0.5 \cdot 10^6$, mit Vektorisierung, (b) $n = 8.0 \cdot 10^6$, mit Vektorisierung, (c) $n = 0.5 \cdot 10^6$, ohne Vektorisierung, (d) $n = 8.0 \cdot 10^6$, ohne Vektorisierung.

n	Beste Blockgröße	Laufzeit der besten Blockgröße (s)	Schlechteste Blockgröße	Laufzeit der schlechtesten Blockgröße (s)
ICC 14.0.1, mit Vektorisierung				
$0.5 \cdot 10^6$	128	0.051	65 536	0.097
$8.0 \cdot 10^6$	128	0.817	1 048 576	2.360
ICC 14.0.1, ohne Vektorisierung				
$0.5 \cdot 10^6$	16	0.061	65 536	0.095
$8.0 \cdot 10^6$	256	0.977	1 048 576	2.393

Tabelle 9.8.: Einfluss der Vektorisierung auf die Laufzeit der besten und der schlechtesten Blockgröße.

Blockgröße nicht der gesamte Suchraum möglicher Blockgrößen durchsucht werden muss, sondern es reicht aus, anhand relevanter Eigenschaften der Cache-Hierarchie des Zielsystems und der Größenberechnung laufzeitrelevanter Arbeitsräume eine kleine Menge potenziell effizienter Blockgrößen zu ermitteln, die dann online mit Hilfe heuristisch-empirischer Suche evaluiert wird.

9.5.2. Vergleich paralleler Implementierungsvarianten im Kandidatenpool

Bevor im nächsten Abschnitt die Laufzeitexperimente für den entwickelten Autotuning-Algorithmus präsentiert werden, wird das Laufzeitverhalten der im Kandidatenpool enthaltenen parallelen Implementierungsvarianten für gemeinsamen Adressraum verglichen. Außerdem wird untersucht, wie die neu entwickelten spezialisierten Implementierungsvarianten mit verteilter Abspeicherung des Argumentvektors $\mathbf{y}$ (Suffix **yl*) im Vergleich zu Varianten mit gemeinsamem Vektor für $\mathbf{y}$ skalieren.

Zur Bewertung der Skalierbarkeit von parallelen Implementierungsvarianten wird der Speedup, $S_p(n)$, betrachtet. Dieser ist definiert als das Verhältnis der Laufzeit der schnellsten sequentiellen Implementierung, $T_s(n)$, und der Laufzeit der parallelen Implementierung bei Verwendung von p Prozessoren zur Lösung eines Problems der Größe n, $T_p(n)$:

$$S_p(n) = \frac{T_s(n)}{T_p(n)}. \tag{9.3}$$

Theoretisch ist der Speedup nach oben durch die Anzahl der Prozessoren p beschränkt:

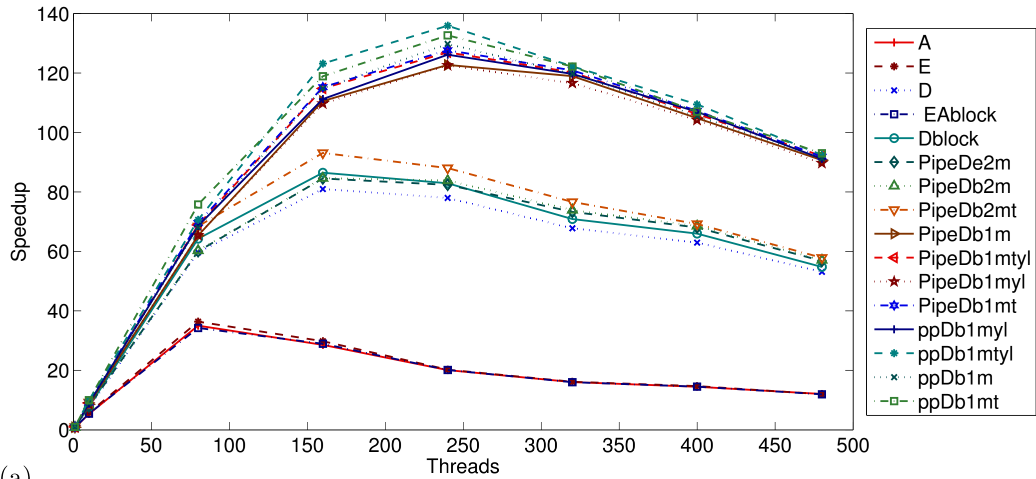
$$S_p(n) \leq p.$$

In der Praxis kann jedoch ein superlinearer Speedup, $S_p(n) > p$, auftreten, der meistens durch Cacheeffekte bedingt ist [RR12a].

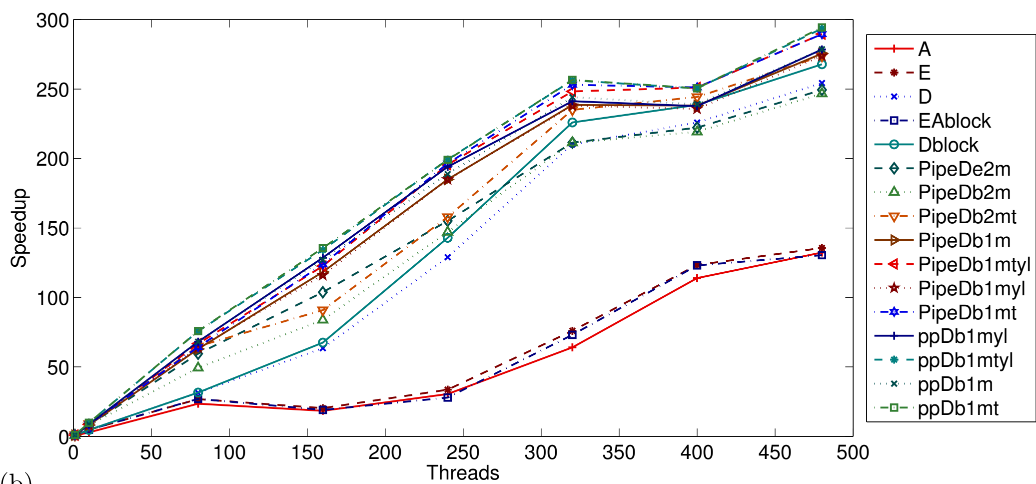
Der theoretisch ideale Speedup ist linear, d. h. $S_p(n) = p$. Erfahrungsgemäß wird dieser aber selten erreicht, da es bei jedem Programm einen sequentiellen Anteil f gibt, der nicht parallelisiert werden kann. Zudem benötigen parallele Implementierungen zusätzliche Zeit u. a. zum Austausch von Daten, zur Synchronisation von Prozessen und für Wartezeiten, die in Folge ungleicher Lastverteilung entstehen [RR12a]. Nimmt man an, dass der nicht parallelisierbare Anteil eines Programms mit der wachsenden Eingabegröße abnimmt, so lässt sich der gewünschte Speedup $\leq p$ durch das Erhöhen der Eingabegröße erreichen [RR12a]. Es gibt jedoch Probleme, die sich nicht beliebig vergrößern lassen.

Die Abbildungen 9.15 (a) und (b) zeigen die von parallelen Implementierungsvarianten auf dem Rechnersystem SGI UV erreichten Speedups für das Testproblem STRING und zwei Systemgrößen $n = 0.5 \cdot 10^6$ und $n = 8.0 \cdot 10^6$. Für Loop-Tiling-Varianten sind die Blockgrößen mit dem im Abschnitt 8.4.2 beschriebenen Blockgrößenauswahlansatz bestimmt worden. Als Referenz für die Speedup- Berechnung paralleler Varianten wurde die Laufzeit der schnellsten sequentiellen Implementierung (durch Autotuning-Algorithmus bestimmt) verwendet.

Für die Problemgröße $n = 0.5 \cdot 10^6$ (Abb. 9.15 (a)) kann man beobachten, dass mit



(a)



(b)

Abbildung 9.15.: Speedups der im Kandidatenpool enthaltenen parallelen Implementierungsvarianten für gemeinsamen Adressraum, gemessen auf der Testplattform SGI UV für das STRING-Problem. (a) $n = 0.5 \cdot 10^6$, (b) $n = 8.0 \cdot 10^6$.

wachsender Anzahl der Threads die spezialisierten Varianten wesentlich schneller laufen als die allgemeinen Varianten. Für die Systemgröße $n = 8 \cdot 10^6$ dagegen erreichen viele allgemeine Varianten auch bei großen Threadzahlen gute Speedups.

Die Hauptursache für eine im Vergleich mit spezialisierten Varianten geringere Skalierbarkeit allgemeiner Varianten bei der Nutzung kleiner Systemgrößen und einer großen Threadzahl sind die hohen Synchronisationskosten, die durch Barrier-Operationen in jedem Korrektorschritt entstehen. Die allgemeinen Implementierungsvarianten A, E und *EAblock* benötigen zwei Barrier-Operationen pro Korrektorschritt und sind daher für große Threadzahlen die langsamsten Varianten. Die übrigen allgemeinen Implementierungsvarianten nutzen nur eine Barrier-Operation pro Korrektorschritt. Sie sind zwar schneller als die Implementierungen A, E und *EAblock*, können aber für die relativ kleine Systemgröße $n = 0.5 \cdot 10^6$ nicht mit spezialisierten Implementierungen konkurrieren, da diese keine Barrier-Operationen, sondern nur eine vergleichsweise schnelle Synchronisation mittels Locks verwenden.

Bei allgemeinen Implementierungsvarianten wird mit wachsender Threadanzahl der erforderliche Berechnungsaufwand pro Thread kleiner, der Synchronisationsaufwand dagegen vergrößert sich. Ist die Anzahl der Gleichungen pro Thread relativ klein, so macht die Laufzeit von Barrier-Operationen nur einen kleinen Teil der Laufzeit eines Zeitschrittes aus. Mit wachsender Threadzahl verteuert sich die Ausführung einer Barrier-Operation und weil der Berechnungsaufwand abnimmt (weniger Gleichungen pro Thread), macht die Zeit für Barrier-Operationen einen größeren Anteil eines Zeitschrittes aus. Beispielsweise beträgt die Laufzeit einer aktiv-wartenden Barrier zur Synchronisation von 400 Threads auf dem SGI UV System $\approx 200\,\mu s$. Demnach benötigt eine einzelne Barrier-Operation für die Implementierung D und die Systemgröße $n = 0.5 \cdot 10^6$ ca. 6 % der Laufzeit eines Zeitschrittes. Wird das Lobatto III C(8)-Verfahren mit 7 Korrektorschritten benutzt, so muss die Implementierung D 7 Barrier-Operationen pro Korrektorschritt ausführen, die sich zusammen zu 42 % der Laufzeit eines Zeitschrittes summieren. Im Gegensatz dazu macht die Zeit zum Ausführen von Barrier-Operationen bei der Variante D für $n = 8.0 \cdot 10^6$ nur $\approx 10\,\%$ der Laufzeit eines Zeitschrittes aus.

9.5.3. Performance des selbstadaptiven Solvers

Die Performance des selbstadaptiven Solvers wurde anhand zweier Testprobleme, STRING und BRUSS2D, auf drei verschiedenen Zielarchitekturen und für unterschiedliche Threadzahlen evaluiert. Diese Ergebnisse sind in den Tabellen 9.9 und 9.10 zusammengefasst.

Da die Vorauswahl- und die Online-Profiling-Phase nur wenig Laufzeit benötigen, soll mit Hilfe der Experimente vor allem untersucht werden, wie lang die Autotuningphase dauert.

In den Tabellen 9.9 und 9.10 sind angegeben (von links nach rechts): die Systemgröße (n), die Threadanzahl (p), die vom Autotuner gewählte Implementierungsvariante und Blockgröße, die Laufzeit der gewählten Implementierungsvariante pro Zeitschritt (TPS) in Sekunden, die benötigte Anzahl der Autotuning-Zeitschritte, die Zeit der Autotuningphase, die Anzahl der in der Autotuningphase evaluierten Implementierungsvarianten, die Gesamtanzahl der Zeitschritte, für die der Autotuning-Overhead $\leq 5\,\%$ beträgt, die

langsamste Implementierungsvariante und ihre Laufzeit pro Zeitschritt (TPS) in Sekunden und der Speedup der ausgewählten Implementierungsvariante zu der langsamsten Variante.

Da eine erschöpfende Suche nach der optimalen Blockgröße für jede mögliche Parameterkombination (Systemgröße, Implementierungsvariante, Korrektorverfahren, Threadzahl) sehr zeitaufwendig wäre, wurde in den Experimenten davon ausgegangen, dass die durch den selbstadaptiven Solver bestimmte Blockgröße optimal ist.

Die Gesamtanzahl der Zeitschritte, für ab der der Autotuning-Overhead $\leq 5\,\%$ beträgt (s. Tab. 9.9, Spalte 9), wird wie folgt berechnet:

$$S_{5\,\%} = \frac{T_{\text{at}}/t_{\text{best}} - S_{\text{at}}}{0.05}. \tag{9.4}$$

Dabei bezeichnet T_{at} die Zeit der Autotuningphase (Spalte 7), S_{at} ist die Anzahl der Autotuning-Zeitschritte und t_{best} ist die Laufzeit pro Zeitschritt der durch den selbstadaptiven Solver bestimmten schnellsten Implementierungsvariante (Spalte 5).

Die Ergebnisse in den Tabellen 9.9, 9.10 und 9.11 zeigen, dass die Anzahl der Autotuning-Schritte zur Auswahl der schnellsten Implementierungsvariante mit einer geeigneten Blockgröße in einem akzeptablen Bereich liegt: auf dem Leo-System sind dafür zwischen 45 und 54 Zeitschritte, auf dem SGI UV zwischen 20 und 61 und auf dem 2-Socket-Nehalem-System zwischen 41 und 54 notwendig. Die Gesamtzahl benötigter Autotuning-Schritte hängt dabei nicht nur von der Anzahl evaluierter Blockgrößen ab, sondern auch von der Anzahl getesteter Implementierungsvarianten.

Weil auf dem SGI UV System eine große Anzahl von Threads gestartet werden kann, gibt es mehrere Testfälle, in denen Implementierungsvarianten anhand der abgeschätzten Synchronisationszeit vorab aussortiert werden können und deshalb nicht auf der Zielplattform evaluiert werden müssen. Beim STRING-Problem mit $n = 0.5 \cdot 10^6$ und Threadanzahl ≥ 320 werden in der Autotuningphase wegen des hohen Synchronisationsaufwandes alle allgemeinen Implementierungsvarianten aussortiert, sodass nur 8 spezialisierte Varianten zur Laufzeit evaluiert werden müssen. Beim BRUSS2D werden für $n = 0.5 \cdot 10^6$ und eine Threadzahl größer als 55 sowie für $n = 8.0 \cdot 10^6$ und einer Threadzahl größer als 222 in der Vorauswahlphase 4 Pipelining-Varianten (mit Prefix *pp*) aus dem Kandidatenpool ausgeschlossen, weil ansonsten nicht ausreichend viele Komponenten pro Thread zum Aufbau einer vollen Pipeline vorhanden sind. In den Experimenten mit kleinerer Systemgröße $n = 0.5 \cdot 10^6$ und Threadzahl ≥ 320 werden in der Autotuningphase zusätzlich die 3 Implementierungsvarianten A, E und *EAblock* wegen des hohen Synchronisationsaufwandes aussortiert. Da auf dem Leo-System, das mit 48 Kernen ausgestattet ist, und dem 2-Socket-Nehalem-System mit 8 Kernen nicht so viele Threads gestartet werden und der Synchronisationsaufwand sehr niedrig bleibt, müssen in der Autotuningphase alle 16 Implementierungsvarianten evaluiert werden.

In den Tabellen 9.9, 9.10 und 9.11 gibt die Spalte "schl. Impl." die langsamste Implementierung an. Die Spalte "TPS Schl. (s)" zeigt für diese Variante die Laufzeit pro Zeitschritt (ausgeführt mit einer guten Blockgröße, die durch die Blockgrößenauswahlstrategie ermittelt wurde). Für das STRING-Problem mit $n = 0.5 \cdot 10^6$ auf dem SGI UV

n	p	gew. Impl.	gew. Blockgröße	gew. TPS (s)	♯Schritte für AT	Zeit für AT (s)	♯eval. Impl.	♯Schritte 5 % Ovhd.	schl. Impl.	TPS Schl. (s)	Speedup Schl./Gew.
BRUSS2D auf Leo											
$0.5 \cdot 10^6$	1	*ppDb1mtyl*	3680	0.823	45	43.984	16	169	*A*	1.410	1.71
$0.5 \cdot 10^6$	6	*ppDb1mt*	664	0.142	47	10.793	16	581	*A*	0.547	3.85
$0.5 \cdot 10^6$	12	*ppDb1mt*	568	0.071	44	4.779	16	467	*D*	0.233	3.28
$0.5 \cdot 10^6$	24	*ppDb1mt*	664	0.037	45	2.321	16	355	*A*	0.109	2.94
$0.5 \cdot 10^6$	48	*ppDb1mt*	280	0.019	46	1.012	16	146	*A*	0.333	17.53
$8.0 \cdot 10^6$	1	*ppDb1mt*	1576	13.838	46	749.199	16	163	*A*	23.897	1.73
$8.0 \cdot 10^6$	6	*PipeDb1mtyl*	728	2.505	50	196.282	16	568	*A*	13.980	5.58
$8.0 \cdot 10^6$	12	*PipeDb1mtyl*	728	1.250	53	111.212	16	720	*A*	6.893	5.51
$8.0 \cdot 10^6$	24	*PipeDb1mtyl*	360	0.636	54	57.102	16	716	*A*	3.474	5.46
$8.0 \cdot 10^6$	48	*PipeDb1mtyl*	360	0.320	51	24.908	16	537	*A*	1.358	4.24
STRING auf Leo											
$0.5 \cdot 10^6$	1	*ppDb1mt*	560	0.357	47	23.958	16	403	*A*	1.017	2.85
$0.5 \cdot 10^6$	6	*ppDb1mtyl*	608	0.064	53	7.124	16	1167	*A*	0.318	4.97
$0.5 \cdot 10^6$	12	*ppDb1mt*	608	0.031	59	4.075	16	1450	*A*	0.215	6.93
$0.5 \cdot 10^6$	24	*ppDb1mt*	560	0.015	51	1.527	16	1016	*A*	0.066	4.40
$0.5 \cdot 10^6$	48	*ppDb1mtyl*	360	0.008	45	0.507	16	409	*A*	0.027	3.37
$8.0 \cdot 10^6$	1	*ppDb1mtyl*	560	5.734	47	393.426	16	433	*A*	16.434	2.87
$8.0 \cdot 10^6$	6	*ppDb1mtyl*	560	1.028	48	174.540	16	2436	*A*	13.612	13.24
$8.0 \cdot 10^6$	12	*ppDb1mtyl*	608	0.512	54	82.070	16	2126	*A*	6.811	13.30
$8.0 \cdot 10^6$	24	*ppDb1mtyl*	560	0.262	50	43.433	16	2316	*A*	3.599	13.74
$8.0 \cdot 10^6$	48	*ppDb1mtyl*	560	0.127	50	18.730	16	1950	*A*	1.237	9.74

Tabelle 9.9.: Experimentelle Evaluierung des Autotuning-Algorithmus anhand der Testprobleme BRUSS2D und STRING auf dem Leo-System.

n	p	gew. Impl.	gew. Blockgröße	gew. TPS (s)	♯Schritte für AT	Zeit für AT (s)	♯eval. Impl.	♯Schritte 5 % Ovhd.	schl. Impl.	TPS Schl. (s)	Speedup Schl./Gew.
BRUSS2D auf SGI UV											
$0.5 \cdot 10^6$	1	*ppDb1mt*	1840	0.453	49	24.457	16	100	*A*	0.584	1.29
$0.5 \cdot 10^6$	10	*ppDb1mt*	1840	0.046	44	2.367	16	150	*A*	0.071	1.54
$0.5 \cdot 10^6$	80	*PipeDb2mt*	1224	0.007	31	0.417	12	560	*A*	0.031	4.43
$0.5 \cdot 10^6$	160	*PipeDb1mt*	208	0.005	30	0.462	12	1400	*A*	0.024	4.80
$0.5 \cdot 10^6$	320	*PipeDb1mt*	208	0.004	24	0.726	9	2953	*E*	0.020	5.00
$0.5 \cdot 10^6$	480	*PipeDb1mt*	208	0.006	20	1.344	9	4308	*A*	0.028	4.67
$8.0 \cdot 10^6$	1	*ppDb1mtyl*	1840	7.312	45	378.633	16	136	*A*	11.485	1.57
$8.0 \cdot 10^6$	10	*PipeDb2mt*	608	0.770	57	55.572	16	304	*A*	2.274	2.95
$8.0 \cdot 10^6$	80	*PipeDb1mtyl*	1224	0.108	61	23.201	16	3077	*D*	1.758	16.28
$8.0 \cdot 10^6$	160	*PipeDb1mt*	1224	0.061	49	6.326	16	1095	*A*	0.674	11.05
$8.0 \cdot 10^6$	320	*PipeDb2mt*	1224	0.029	52	4.499	12	2063	*A*	0.256	8.83
$8.0 \cdot 10^6$	480	*PipeDb2mt*	1840	0.028	43	5.012	12	2720	*A*	0.174	6.21
STRING auf SGI UV											
$0.5 \cdot 10^6$	1	*ppDb1mtyl*	1840	0.308	49	16.956	16	122	*A*	0.439	1.42
$0.5 \cdot 10^6$	10	*ppDb1mt*	456	0.030	45	1.688	16	207	*A*	0.058	1.94
$0.5 \cdot 10^6$	80	*ppDb1mt*	280	0.004	46	0.392	16	974	*A*	0.024	6.00
$0.5 \cdot 10^6$	160	*ppDb1mtyl*	304	0.002	46	0.362	16	1954	*A*	0.011	5.50
$0.5 \cdot 10^6$	320	*ppDb1mtyl*	280	0.002	28	0.645	8	4953	*E*	0.012	6.00
$0.5 \cdot 10^6$	480	*ppDb1mt*	456	0.003	29	1.230	8	6720	*EAblock*	0.028	9.33
$8.0 \cdot 10^6$	1	*ppDb1mt*	1840	4.890	48	275.886	16	169	*A*	9.186	1.88
$8.0 \cdot 10^6$	10	*ppDb1mtyl*	1840	0.501	51	34.979	16	391	*A*	2.115	4.26
$8.0 \cdot 10^6$	80	*ppDb1mtyl*	456	0.064	45	5.151	16	710	*A*	0.530	8.28
$8.0 \cdot 10^6$	160	*ppDb1mt*	456	0.034	53	7.477	16	3339	*A*	0.780	22.94
$8.0 \cdot 10^6$	320	*ppDb1mtyl*	456	0.017	50	2.444	16	1876	*A*	0.271	15.94
$8.0 \cdot 10^6$	480	*PipeDb2mt*	64	0.016	48	2.732	16	2455	*A*	0.171	10.69

Tabelle 9.10.: Experimentelle Evaluierung des Autotuning-Algorithmus auf der SGI UV für die Testprobleme BRUSS2D und STRING.

n	p	gew. Impl.	gew. Blockgröße	gew. TPS (s)	♯Schritte für AT	Zeit für AT (s)	♯eval. Impl.	♯Schritte 5 % Ovhd.	schl. Impl.	TPS Schl. (s)	Speedup Schl./Gew.
BRUSS2D auf 2-Socket-Nehalem											
$0.5 \cdot 10^6$	1	*PipeDb2mt*	1224	0.392	47	19.493	16	55	*A*	0.475	1.21
$0.5 \cdot 10^6$	8	*ppDb1mt*	1496	0.052	41	2.445	16	121	*A*	0.094	1.81
$8.0 \cdot 10^6$	1	*PipeDb2mt*	1224	6.272	47	314.233	16	63	*PipeDe2m*	7.693	1.23
$8.0 \cdot 10^6$	8	*PipeDb2mt*	1224	0.835	46	45.712	16	175	*A*	1.912	2.90
STRING auf 2-Socket-Nehalem											
$0.5 \cdot 10^6$	1	*PipeDb2mt*	1224	0.245	54	14.473	16	102	*A*	0.317	1.29
$0.5 \cdot 10^6$	8	*ppDb1mtyl*	664	0.033	52	2.069	16	214	*A*	0.075	2.28
$8.0 \cdot 10^6$	1	*PipeDb2mt*	1224	3.913	52	221.375	16	92	*A*	5.110	1.31
$8.0 \cdot 10^6$	8	*ppDb1mtyl*	1840	0.529	50	33.717	16	275	*A*	1.648	3.11

Tabelle 9.11.: Experimentelle Evaluierung des Autotuning-Algorithmus auf dem 2-Socket-Nehalem-System für die Testprobleme BRUSS2D und STRING.

mit 480 Threads ist die langsamste Implementierungsvariante *EAblock* mit Blockgröße 152 (ausgewählt durch die Blockgrößenauswahlstrategie). Für alle anderen Benchmarks ist eine der drei Varianten A, E oder D die Langsamste.

Die Tabellen zeigen auch die Leistungsverbesserung der ausgewählten Implementierungsvariante im Vergleich zu der langsamsten Variante (Spalte "Speedup Schl./Gew."). Für die in den Tabellen präsentierten Benchmarks ist die gewählte Implementierungsvariante 1.21-22.94-mal schneller als die langsamste Variante.

Der zusätzliche Zeitaufwand zum Durchführen des Autotunings ist nicht nur von der Anzahl der Autotuning-Zeitschritte und ihrer Dauer abhängig, sondern auch von der Gesamtzahl der Zeitschritte und der Gesamtberechnungszeit, die zur Lösung des gegebenen Anfangswertproblems mit der besten Variante erforderlich sind. Je kleiner die Zeit pro Schritt der besten Implementierungsvariante und je höher die Autotuning-Zeit, umso mehr Zeitschritte müssen ausgeführt werden, um die Dauer des Autotunings zu kompensieren.

Die Spalte "♯Schritte 5 % Ovhd." gibt die Anzahl der Schritte, für die der Autotuning-Overhead 5 %. beträgt (s. Gleichung (9.4)). Diese hängt von der Laufzeit pro Zeitschritt der gewählten Implementierungsvariante, der Anzahl der Autotuning-Zeitschritte und der Dauer des Autotunings ab. In den Experimenten variiert ihre Anzahl zwischen 55 und 6720. Da bei echten numerischen Simulationen die Lösung eines Anfangswertproblems nicht selten mehrere Hunderttausende bis Millionen Zeitschritte erfordert, ist der Autotuning-Overhead verschwindend gering.

9.6. Zusammenfassung

In diesem Kapitel wurden zunächst die zur Leistungsverbesserung vorgenommenen Änderungen an den bereits existierenden parallelen Implementierungsvarianten aus [KR07b] und neue parallele Implementierungsvarianten mit verteilter Abspeicherung des Argumentvektors $\mathbf{y}$ beschrieben. Danach wurde der selbstadaptive parallele ODE-Solver für gemeinsamen Adressraum präsentiert. Die geänderte Strategie der Blockgrößenauswahl wurde dargestellt und eine Laufzeitvorhersage zum Ausschluss langsamer Implementierungsvarianten aus dem Kandidatenpool beschrieben.

Die Effizienz der Blockgrößenauswahlstrategie wurde auf verschiedenen Rechnersystemen, für unterschiedliche Implementierungsvarianten, Systemgrößen und Korrektorverfahren untersucht. Ferner wurde untersucht, wie wirkt sich die Vektorisierung auf die Wahl von Blockgrößen auswirkt.

Die durchgeführten Experimente zeigen, dass zur Auswahl einer geeigneten Blockgröße nicht der gesamte Suchraum möglicher Blockgrößen durchsucht werden muss, sondern dass es ausreicht, anhand relevanter Eigenschaften der Cache-Hierarchie des Zielsystems und der Größenberechnung laufzeitrelevanter Arbeitsräume eine kleine Menge potenziell effizienter Blockgrößen zu ermitteln, die dann online mit Hilfe heuristisch-empirischer Suche evaluiert werden. In allen Testfällen war die Anzahl der zur Laufzeit evaluierten Blockgrößen sehr klein. Diese Anzahl reichte jedoch aus, um effiziente Blockgrößen bestimmen zu können. Die Performance gewählter Blockgrößen lag im Durchschnitt 2-3 %

von der optimalen Blockgröße entfernt. Zudem haben die Experimente gezeigt, dass, wenn Vektorisierung verwendet wird, die Wahl einer effizienten Blockgröße besonders wichtig ist.

Beim Autotuning-Algorithmus benötigen die Phasen der Vorauswahl und des Online-Profilings nur wenig Laufzeit, der Autotuning-Overhead wird im wesentlichen durch die Dauer der Autotuningphase bestimmt. Der Autotuning-Overhead hängt von der Anzahl der Autotuning-Zeitschritte und ihrer Dauer sowie von der Gesamtberechnungszeit, die zur Lösung des gegebenen Anfangswertproblems mit der besten Variante erforderlich ist, ab. Je kleiner die Zeit pro Schritt der besten Implementierungsvariante und je höher die benötigte Autotuning-Zeit, umso mehr Zeitschritte müssen ausgeführt werden, um die Dauer des Autotunings zu kompensieren. In Experimenten wurde die Gesamtanzahl der Schritte berechnet, die ausgeführt werden müssen, damit der Autotuning-Overhead nur 5 % beträgt. Ihre Anzahl lag zwischen 55 und 6720, abhängig von der Laufzeit der gewählten Implementierungsvariante, der Anzahl der Autotuning-Zeitschritte und der Dauer der Autotuningphase.

10. Weitere durchgeführte und laufende Arbeiten zur Erweiterung des Autotuning-Frameworks

In diesem Kapitel werden weitere durchgeführte und laufende Arbeiten zur Erweiterung des Autotuning-Frameworks präsentiert.

10.1. Automatische Generierung von Implementierungsvarianten

Infolge der vorangegangenen Arbeit von Matthias Korch [KR07b] stand ein großer Pool sequentieller bzw. paralleler Implementierungsvarianten des betrachteten PC-Verfahrens für gemeinsamen Adressraum zur Verfügung, der vom Autotuner genutzt werden konnte. Damit jedoch die in dieser Arbeit vorgestellten Techniken auch auf andere ODE-Zeitschrittverfahren angewendet und darüber hinaus noch weitere Optimierungstechniken, wie z. B. Loop-Unrolling oder Scalar-Replacement, untersucht werden können, wäre eine Integration eines Transformationswerkzeugs wünschenswert, das durch die Anwendung unterschiedlicher Schleifentransformationen die Varianten automatisch aus einer Basisvariante ableiten kann.

Es lassen sich nicht alle Implementierungsvarianten durch Anwendung von Schleifentransformationen herleiten. Beispielsweise kann die Implementierung E durch Permutation der beiden inneren Schleifen aus der Implementierung A generiert werden, jedoch kann die Implementierung D nicht aus A erzeugt werden, da sie unterschiedliche Datenstrukturen verwendet. Es wäre allerdings möglich, eine Reihe grundlegender Implementierungsvarianten, die u. U. verschiedene Datenstrukturen nutzen, aufzustellen, aus denen dann weitere Varianten mithilfe eines Source-to-Source-Compilers erzeugt werden.

Der Einsatz von Übersetzerwerkzeugen zur Generierung von Implementierungsvarianten war das Thema des Masterprojekts von Johannes Seifert. Im Rahmen dieses Projekts ist ein Transformationswerkzeug namens *PALT (Pragma Assisted Loop Transformations)* entstanden, das gewünschte Schleifentransformationen auf einen vorgegebenen C/C++ Quellcode anwenden kann. PALT unterstützt folgende Schleifentransformationen: Loop-Fission, Loop-Fusion, Loop-Interchange, Loop-Tiling und Loop-Unrolling. Zum Durchführen dieser Codetransformationen nutzt PALT den ROSE-Compiler [QL11] zurück. Die Liste von Transformationen, die nacheinander auf die gewünschte Schleife oder einen Schleifenkomplex angewendet werden sollen, werden in einer XML-basierten Beschreibung spezifiziert. Der zu transformierende Quellcode wird mithilfe des ROSE-Compilers in Form eines Abstrakten Syntaxbaumes (AST) dargestellt. PALT parst die XML-Beschreibung gewünschter Transformationen und führt diese auf dem AST aus. Am Ende wird aus der geänderten AST-Darstellung der entsprechende Quellcode generiert.

10.2. Automatische Bestimmung der Anzahl von Referenzen

Beim Autotuning-Ansatz für sequentielle bzw. parallele Implementierungsvarianten wurden die Arbeitsraumformeln zur Bestimmung der Anzahl innerhalb eines Arbeitsraumes referenzierten Datenelemente manuell aufgestellt (vgl. Abschnitt 8.4.2 und 9.4.2). Hilfreich wäre es jedoch, wenn die Anzahl der Referenzen innerhalb eines Arbeitsraumes automatisch erfasst werden könnte. Im aktuellen Masterprojekt von Daniel Heinlein soll die Anzahl der innerhalb eines Arbeitsraumes referenzierten Datenelemente automatisch bestimmt werden. Die Idee dabei ist es, eine Menge diskreter Punkte (Stützstellen) zu wählen, an denen die Anzahl der Referenzen tatsächlich bestimmt wird, und die Arbeitsraumfunktion durch eine stetige Funktion zu approximieren. Die Approximation erfolgt durch eine Polynomausgleichsrechnung. Dabei wird nach einem multivariaten Polynom in n Variablen $p(x) = \sum_{\alpha} a_\alpha x^\alpha$ (mit Monomen $x^\alpha = \prod_{j=1}^{n} x_j^{\alpha_j}$ und $\alpha = (\alpha_1, \ldots, \alpha_n) \in \mathbb{N}_0^n$) gesucht, das die D Stützpunkte $(\bar{x}_1, f(\bar{x}_1)), \ldots, (\bar{x}_D, f(\bar{x}_D))$ möglichst gut annähert. Um das Ausgleichspolynom zu bestimmen, wird die Methode der kleinsten Quadrate [Gau06] verwendet.

10.3. Ein selbstadaptiver paralleler ODE-Solver für verteilten Adressraum und Optimierung des Energieverbrauchs

Beim automatischen Software-Tuning wird das Programm automatisch bezüglich eines oder mehrerer Ziele optimiert (s. Kapitel 4.3). Das Ziel dabei muss nicht immer die Minimierung der Ausführungszeit sein. Aufgrund steigender Energiekosten kann eine Programmoptimierung bezüglich der verbrauchten Energie oder eines Kompromisses zwischen der Laufzeit und der Energie, als Energy-Delay-Produkt (EDT) [HIG94] bezeichnet, von Interesse sein. Das Energy-Delay-Produkt ist das Produkt der umgewandelten Energie und der Laufzeit eines Programms. Anstelle des EDTs kann auch das Energy-Delay-Square-Produkt [Mar01, PM02] als Metrik verwendet werden. Bei dieser ist die Laufzeit im Vergleich zur Energie höher gewichtet.

Die Prozessoren der Firma Intel und AMD sind seit neuestem mit Sensoren ausgestattet, die es erlauben den Energieverbrauch, ohne Anschluss zusätzlicher Hardware zu ermitteln. Intel-Prozessoren verfügen seit der Sandy-Bridge-Generation über die **Running-Average-Power-Limit (RAPL)**-Technologie zum Ermitteln des Energieverbrauchs einer CPU. PAPL wurde dazu entwickelt, die Leistungsaufnahme einer CPU zu überwachen und bietet die Möglichkeit eine Obergrenze für die Leistungsaufnahme einer CPU festzulegen [Lan14]. Der Energieverbrauch wird nicht direkt gemessen, sondern mithilfe eines Software-Leistungsmodells (engl. Software Power Model), das u. a. Hardware-Performance-Zähler nutzt, abgeschätzt [RNA+12]. Das RAPL-Interface erlaubt die Messung der Leistungsaufnahme verschiedener Komponenten (Domänen) einer CPU. Welche Domänen verfügbar sind, hängt von der Prozessor-Version [HIS+13] ab. Die abgeschätzten Werte für den Energieverbrauch seit einem bestimmten Zeitpunkt werden in maschinenspezifischen Registern (MSRs) gespeichert [Lan14]. AMD-Prozesso-

ren ab der Bulldozer-Generation verfügen über eine ähnliche Technik zur Beschränkung der Leistungsaufnahme, die als **Application-Power-Management (APM)** bezeichnet wird [AMDA13]. Die Werte für den Energieverbrauch können jedoch nicht direkt aus den MSR-Registern, sondern nur über einen Sideband-Interface-Bus gelesen werden [Lan14].

Im Rahmen der Masterarbeit von Frank Wein [Wei14] wurde ein selbstadaptiver ODE-Solver für verteilten Adressraum entwickelt. Dieser unterstützt verschiedene Optimierungsmöglichkeiten: die beste Implementierungsvariante kann bezüglich der Laufzeit, des Energieverbrauchs oder des Energy-Delay-Produkts gewählt werden.

Die Auswahlmenge von Implementierungsvarianten für den Autotuner besteht aus in [Str12] entwickelten parallelen Varianten für verteilten Adressraum, die sich in der Schleifenreihenfolge, den Datenstrukturen und verwendeten Kommunikationsmustern unterscheiden. Darunter sind allgemeine Varianten, die auf alle ODE-Probleme anwendbar sind und Multibroadcastoperationen wie *MPI_Allgatherv* zur Prozess-Synchronisation in jedem Zeitschritt verwenden (A-Varianten), Varianten mit angepasster Kommunikation, die speziell für dünnbesetzte DGLs entwickelt worden sind (S-Varianten), Varianten mit Nachbarschaftskommunikation (N- und X-Varianten), die nur bei beschränkter Zugriffsdistanz des ODE-Problems eingesetzt werden können. Der Unterschied zwischen N- und X-Varianten besteht darin dass, dass bei X-Varianten die Daten auch über mehrere Nachbarprozesse weiter versendet werden können und diese deshalb auch bei Datenverteilungen funktionieren, bei denen die Anzahl der Komponenten pro Thread größer oder gleich der Zugriffsdistanz $d(\mathbf{f})$ ist.

Zur Auswahl von Blockgrößen für Loop-Tiling-Varianten wurde die modellbasierte Blockgrößenauswahlstrategie aus Kapitel 8.4.2 verwendet. Der implementierte Autotuner kann unterschiedliche CPU-Frequenzen (innerhalb des für den Prozessor zulässigen Bereichs) zur Laufzeit testen und die beste CPU-Frequenz für jede Implementierungsvariante in Hinblick auf das Optimierungsziel bestimmen. Zusätzlich bietet der Autotuner die Möglichkeit der automatischen Anpassung der Anzahl parallel rechnender Prozesse zur Minimierung des Energieverbrauchs bzw. der zur Lösung benötigten Laufzeit.

Die experimentelle Untersuchung des Autotuners wurde auf zwei Testsystemen mit Intel Ivy-Bridge-Architektur, die mit einen i7-3770- und einem i5-3550-Prozessor ausgestattet waren, durchgeführt (für die Beschreibung dieser Systeme siehe [Wei14]). Als Testprobleme wurden BRUSS2D und STRING verwendet. Die Experimente haben gezeigt, dass es vor allem von der Implementierungsvariante abhängt, mit welcher Prozessoranzahl der geringste Energieverbrauch bzw. die beste Performance erreicht wird. Implementierungsvarianten, die Nachbarschaftskommunikation verwenden, wie N- und X-Varianten, sind anderen Varianten beim Energieverbrauch und Laufzeit überlegen und profitieren zusätzlich von Hyperthreading. In Experimenten bewirkte die Änderung der CPU-Frequenz keine Änderung in der Reihenfolge von Implementierungsvarianten bezüglich der Laufzeit und des Energieverbrauchs. Die Messergebnisse machten außerdem deutlich, dass eine schnellere Laufzeit meist zu einem geringeren Energieverbrauch führt.

11. Zusammenfassung und Ausblick

Dieses Kapitel fasst die Dissertation zusammen und gibt einen Ausblick auf mögliche zukünftige Forschungsaufgaben.

11.1. Zusammenfassung

In dieser Arbeit wurden verschiedene Autotuning-Ansätze zum effizienten numerischen Lösen von Anfangswertproblemen gewöhnlicher Differentialgleichungssysteme auf aktuellen Prozessor- und Rechnerarchitekturen präsentiert. Die entworfenen Algorithmen erlauben eine automatische Anpassung von ODE-Verfahren an spezifische Eigenschaften der Zielplattform und an Charakteristika des zu lösenden Anfangswertproblems, insbesondere die Systemgröße und das Zugriffsmuster der Funktion der rechten Seite. Die Anwendbarkeit von Autotuning-Techniken auf ODE-Verfahren wurde anhand einer Klasse expliziter PC-Verfahren vom RK-Typ gezeigt. Diese Verfahren wurden aufgrund ihres großen Parallelitätspotenzials bezüglich des Systems und der Methode gewählt und weil ihre Berechnungsvorschrift die Erzeugung einer Vielzahl von Implementierungsvarianten erlaubt.

Als erstes ist in dieser Arbeit die Anwendbarkeit des Online-Autotunings auf sequentielle Varianten des betrachteten PC-Verfahrens anhand einer Reihe von Experimenten auf unterschiedlichen Rechnersystemen und für verschiedene Anfangswertprobleme gezeigt worden. Daraus entstand ein Grundalgorithmus des Autotunings, der zunächst nur die Auswahl einer guten Implementierungsvariante zur Laufzeit realisiert hat und die Programmparameter, wie z. B. die Blockgröße, unberücksichtigt ließ. Der resultierende Algorithmus klassifiziert die zu lösenden ODE-Systeme anhand ihrer Klassenzugehörigkeit zu Problemen mit beschränkter oder unbeschränkter Zugriffsdistanz und teilt sie in verschiedene Gruppen auf. Zu jeder Gruppe stellt der Algorithmus eine Menge von Implementierungsvarianten, Kandidatenpool genannt, bereit. Der Vergleich von Implementierungsvarianten findet zur Laufzeit statt, während der ersten Zeitschritte des Integrationsverfahrens. Die beste Implementierungsvariante aus dem Kandidatenpool wird anhand eines empirischen Vergleichs der Laufzeiten gewählt.

Im Kandidatenpool sind unter anderen Implementierungsvarianten enthalten, die durch Anwendung von Loop-Tiling entstanden sind und für die eine geeignete Blockgröße bestimmt werden soll. In Experimenten wurde zunächst untersucht, wie stark die Performance einer sequentiellen Loop-Tiling-Variante von der gewählten Blockgröße beeinflusst wird. Für das BRUSS2D-Problem wurde bei Verwendung von Blockgrößen im Bereich $[1, 5000]$ auf verschiedenen Rechnersystemen ein Laufzeitunterschied von maximal 10 % beobachtet. Bei sehr großen Blockgrößen, die nah an die Systemgröße n heran reichen, kam es jedoch zu einem Performanceverlust von mehr als 40 %. Zudem haben die Ex-

perimente gezeigt, dass der Suchraum mehrere Bereiche mit guten Blockgrößen enthält. Innerhalb dieser Bereiche betrug der Laufzeitunterschied zwischen guten und weniger guten Blockgrößen nur einige wenige Prozent. Da es zum Erreichen einer hohen Performance ausreichen ist, eine Blockgröße innerhalb dieser guten Bereichen zu bestimmen, hat die Methode der automatischen Auswahl von Blockgrößen das Ziel, eine gute Blockgröße in möglichst kurzer Zeit zu finden. Durch diese Zielsetzung hält man die Anzahl der zur Laufzeit evaluierten Blockgrößen klein und reduziert somit die Zeit zur Durchführung des Autotunings.

Die entsprechend dieser Zielsetzung entwickelte Strategie zur automatischen Auswahl von Blockgrößen schränkt den Suchraum potentieller Blockgrößen mit Hilfe eines analytischen Modells ein. Dadurch entsteht eine weitaus kleinere Menge von Blockgrößen, die dann zur Laufzeit empirisch evaluiert wird. Das Modell basiert auf der Bestimmung von Arbeitsräumen der in der Implementierungsvariante auftretenden Schleifen und bezieht die relevanten Eigenschaften der Speicherhierarchie der Zielplattform in die Vorauswahl potentiell guter Blockgrößen mit ein. Zusätzlich verwendet das Modell eine auf praktischer Erfahrung basierende Heuristik.

In nächsten Schritt ist die Blockgrößenauswahlstrategie in den Autotuning-Grundalgorithmus für sequentiellen Varianten integriert worden. Die Qualität der vom Modell vorausgewählten Blockgrößen in Bezug auf die Laufzeit der optimalen Blockgröße wurde systematisch auf drei verschiedenen Rechnersystemen und für unterschiedliche Implementierungsvarianten und Systemgrößen bestimmt. Die Ergebnisse haben gezeigt, dass das Modell präzise genug ist, um gute Blockgrößen vorschlagen zu können. In Experimenten lag mindestens eine der vom Modell vorgeschlagenen Blockgrößen in einem guten Bereich des Suchraumes.

Als nächstes erfolgte die Übertragung der sequentiellen Autotuning-Konzepte auf parallele Implementierungsvarianten mit gemeinsamem Adressraum. Dabei ist die Blockgrößenauswahlstrategie so erweitert worden, dass eine größere Zahl von Blockgrößen mithilfe des Modells vorselektiert wird, indem alle Kombinationen aus Arbeitsräumen und Cachestufen berücksichtigt werden. Die beste Blockgröße wird dann mit Hilfe einer heuristisch-empirischen Suche aus der vorselektierten Menge gewählt, wobei trotz der Vergrößerung der Vorauswahlmenge die Zahl empirisch evaluierter Blockgrößen klein gehalten werden kann. Die Effizienz der Blockgrößenauswahlstrategie wurde auf verschiedenen Rechnersystemen, für unterschiedliche Implementierungsvarianten, Systemgrößen und Korrektorverfahren evaluiert. Ferner wurde untersucht, wie sich die Vektorisierung auf die Wahl von Blockgrößen auswirkt. Die experimentelle Untersuchung hat gezeigt, dass bei parallelen Varianten die Wahl einer ungünstigen Blockgröße einen wesentlich größeren Einfluss auf die Laufzeit hat, als dies bei sequentiellen Varianten der Fall ist. Nicht nur die Wahl großer Blockgrößen nah an n, sondern auch die Wahl kleiner Blockgrößen kann die Performance paralleler Implementierungsvarianten stark verschlechtern. Zudem wurde deutlich, dass, wenn Vektorisierung verwendet wird, die Wahl einer effizienten Blockgröße besonders wichtig ist. Außerdem haben die Experimente gezeigt, dass die Blockgrößenauswahlstrategie in der Lage ist, effiziente Blockgrößen für unterschiedliche Implementierungsvarianten, zu lösende AWPs und Zielplattformen zu bestimmen

und dass diese ferner nur geringen Zeitaufwand erfordert, da nur einige wenige Blockgrößen zur Laufzeit evaluiert werden müssen. Die Performance der gewählten Blockgrößen war im Durchschnitt 2-3 % schlechter als die der optimalen Blockgröße.

Der entwickelte parallele Autotuning-Algorithmus für gemeinsamen Adressraum enthält eine Strategie zur Reduktion der Anzahl zu evaluierender Implementierungsvarianten. Diese basiert auf der Abschätzung des Zeitanteils für die Synchronisation von Threads.

Im Rahmen einer Masterarbeit wurde der Autotuning-Ansatz für verteilten Adressraum erweitert. Darüber hinaus wurden zusätzliche Optimierungsmöglichkeiten integriert: die beste Implementierungsvariante kann bezüglich der Laufzeit, des Energieverbrauchs oder des Energy-Delay-Produkts gewählt werden.

Im Rahmen einer studentischen Arbeit ist ein Compilerwerkzeug zur automatische Generierung von Implementierungsvarianten entstanden. Außerdem befasste sich ein aktuelles Masterprojekt mit der Implementierung einer Methode zur automatischen Bestimmung der Anzahl der innerhalb eines Arbeitsraumes referenzierten Datenelementen. Mit der Verfügbarkeit dieser Methode müssen die Arbeitsraumformeln für das analytische Modell zur Auswahl von Blockgrößen nicht mehr von Hand aufgestellt werden.

Insgesamt hat die Arbeit gezeigt, dass durch die Verwendung der entworfenen Autotuning-Algorithmen die Zeit zur Lösung eines AWPs für gewöhnliche Differentialgleichungssysteme deutlich verkürzt werden kann. Mit dem Ansatz des Online-Autotunings, der durch Offline-Tests unterstützt wird, können zur Laufzeit geeignete Programmparameter und eine schnelle Implementierungsvariante gewählt werden. Der Autotuning-Overhead hängt generell von der Gesamtzahl der Zeitschritte zur Lösung eines Anfangswertproblems, der Anzahl der Implementierungsvarianten im Kandidatenpool und der Anzahl der zur Laufzeit evaluierten Parametern ab. Die Laufzeitexperimente zeigten jedoch, dass für den betrachteten Kandidatenpool die Anzahl der Zeitschritte und der Zeitaufwand zur Auswahl einer schnellen Implementierungsvariante mit guter Blockgröße gering ist. Bei echten Simulationen kann die Gesamtanzahl der Zeitschritte zur Lösung eines AWPs im Bereich von mehreren Hunderttausenden bis Millionen liegen; der Zeitaufwand zum Durchführen des Autotunings ist dann entsprechend klein.

11.2. Ausblick

Wie bereits in der Zusammenfassung dargestellt, sind in dieser Arbeit verschiedene Autotuning-Algorithmen für die sequentielle und die parallele Ausführung einer Klasse expliziter PC-Verfahren vom RK-Typ entworfen und untersucht worden. Durch den Einsatz dieser Algorithmen kann die Lösung von Anfangswertproblemen gewöhnlicher Differentialgleichungssysteme beschleunigt werden.

Als nächstes werden einige mögliche Erweiterungen der präsentierten Autotunung-Ansätze sowie neue Ideen vorgestellt, die Gegenstand zukünftiger Forschung sein können.

Verbesserung des parallelen Autotuning-Algorithmus für verteilten Adressraum. Der parallele Autotuning-Algorithmus für verteilten Adressraum in [Wei14] soll verbessert

werden. Der Algorithmus testet zur Laufzeit alle möglichen Kombinationen aus CPU-Frequenzen, Implementierungsvarianten und Anzahl der Prozessoren und soll deshalb durch Techniken zur Minimierung des Suchraumes verbessert werden. Da die Laufzeit und der Energieverbrauch von Implementierungsvarianten maßgeblich vom Kommunikationsaufwand abhängen, wäre ein Ausschluss langsamer Varianten anhand der Abschätzung des Zeitanteils für die Kommunikation denkbar.

Übertragbarkeit der entwickelten Autotuning-Techniken auf andere ODE-Verfahrensklassen. Die in dieser Arbeit präsentierten Autotuning-Ansätze für explizite PC-Verfahren vom RK-Typ können in zukünftigen Arbeiten für andere ODE-Verfahrensklassen, wie z. B. Extrapolationsverfahren und implizite RK-Verfahren (z. B. DIRK- [Ale77] oder PDIRK-Verfahren [vdHSC92, RR95, RR99, IN90]), erweitert und eingesetzt werden.

Erweiterung des Kandidatenpools. Der in dieser Arbeit betrachtete Kandidatenpool besteht aus sequentiellen bzw. parallelen Implementierungsvarianten, die durch Schleifenvertauschung und Schleifenverschmelzung, Loop-Tiling und Überlappen von Datenstrukturen entstanden sind. Im Fall von parallelen Implementierungsvarianten enthält der Kandidatenpool zusätzlich Varianten mit unterschiedlichen Synchronisations- bzw. Kommunikationsstrategien und in [Wei14] darüber hinaus spezielle auf dünnbesetzte ODE-Systeme optimierte Varianten. In Zukunft kann der Kandidatenpool durch weitere Varianten ergänzt werden:

- *Varianten mit Loop-Unrolling und Scalar-Replacement.* Die Optimierungstechnik des Loop-Unrollings kann verwendet werden, um Varianten mit parametrisierten Aufrollfaktor zu generieren. Der geeignete Aufrollfaktor soll im Laufe des Autotuning-Prozesses bestimmt werden. Da die Anzahl der Stufen s eines Korrektorverfahrens i. d. R. klein ist, können beispielsweise Schleifen, die über s iterieren, komplett aufgerollt werden. Aber auch ein teilweises Aufrollen kleiner Schleifen innerhalb der Systemdimensionschleifen kann eine Performance-Verbesserung bewirken. Anders als die Blockgröße für Loop-Tiling muss der Aufrollfaktor zur Compile-Zeit bekannt sein, damit die entsprechende Variante erzeugt werden kann. Damit die Aufrollfaktoren zur Laufzeit bestimmt werden können, kann auf Just-in-time-Kompilierung zurückgegriffen werden. Bei diesem Ansatz würde man Varianten mit verschiedenen Aufrollfaktoren während der Autotuningphase parallel auf freien Rechenknoten generieren.

  Im Kombination mit Loop-Unrolling kann die Optimierungstechnik Scalar-Replacement genutzt werden. Bei dieser werden Array-Referenzen, die auf die gleiche Speicheradresse zeigen und auf die häufig zugegriffen wird, durch eine skalare Variable ersetzt. Diese Variable kann in einem Register gespeichert werden, wodurch die Anzahl der Zugriffe auf den Speicher reduziert wird.

- *Explizite Vektorisierung.* Die in dieser Arbeit präsentierten Autotuning-Algorithmen überlassen die Aufgabe der Vektorisierung des Programmcodes dem verwende-

ten Compiler. Die durchgeführten Experimente haben leider gezeigt, dass Compiler nicht immer in der Lage sind den Code automatisch zu vektorisieren und oft zumindest unterstützende Hinweise in Form von Pragmas brauchen, um verktorisierbare Schleifen zu erkennen. Als zukünftige Aufgabe, wäre deshalb der Entwurf eines Source-zu-Source-Transformationswerkzeuges interessant mit dem explizit vektorisierte Varianten für SIMD-Erweiterungen (SSE und AVX) automatisch generiert und in den Kandidatenpool aufgenommen werden können.

- *Parallele Pipelining-Varianten mit reduziertem Speicherplatzbedarf.* Im Rahmen dieser Dissertation wurden für PC-Verfahren vom RK-Typ bereits sequentielle Pipelining-Varianten mit reduziertem Speicherplatzbedarf entworfen. Durch die Überlappung der Argumentvektoren $\mathbf{Y}_l^{(k)}$ über die Korrektorschritte $k = 1, \ldots, m$ und die Stufen $l = 1, \ldots, s$ kann der Speicherplatzbedarf der Pipelining-Varianten von $(3 + s)n + \Theta(sm \cdot d(\mathbf{f}))$ auf $2n + \Theta(sm \cdot d(\mathbf{f}))$ reduziert werden. In der Zukunft soll der Kandidatenpool um parallele Pipelining-Varianten mit reduziertem Speicherplatzbedarf für gemeinsamen und verteilten Adressraum erweitert werden.

ODE-Systeme mit zeitvariantem Berechnungsverhalten der Funktion der rechten Seite. Die in dieser Arbeit präsentierten Autotuning-Algorithmen basieren auf der Annahme, dass die Laufzeit und das Zugriffsmuster der Funktion der rechten Seite $\mathbf{f}$ sich während der Integration nicht verändert, d. h. in jedem Zeitschritt t werden die gleichen Berechnungen durchgeführt und alle Zeitschritte haben annähernd die gleiche Laufzeit. Diese Annahme trifft auf alle in dieser Arbeit betrachteten AWPs zu. Bei manchen AWPs kann sich jedoch das Berechnungsverhalten der Funktion der rechten Seite $\mathbf{f}$ mit der Zeit t ändern. Zum Beispiel könnte eine zusätzliche Substanz zu einer bereits ablaufenden chemischen Reaktion (BRUSS2D) hinzugefügt werden. In zukünftigen Arbeiten sollten die Autotuning-Algorithmen auch die Lösung solcher AWPs unterstützen, z. B. durch ein erneutes Anstoßen des Autotuning-Prozesses.

Autotuning auf GPUs und heterogenen Plattformen. In den letzten Jahren werden zunehmend Grafikprozessoren oder andere Coprozessoren in Supercomputern zur Beschleunigung berechnungsintensiver Aufgaben eingesetzt. In zukünftigen Arbeiten können die Erfahrungen dieser Dissertation dazu genutzt werden, Autotuning-Techniken zur Lösung von AWPs für gewöhnliche Differentialgleichungssysteme auf GPUs und heterogenen Rechnersystemen zu entwickeln.

Anhang

Anhang A.

Verwendete Testprobleme

A.1. Reaktions-Diffusion-Modell Brusselator (BRUSS2D)

Das Brusselator-Problem ist ein klassisches Reaktions-Diffusion-Modell, das eine chemische Reaktion von zwei Substanzen mit oszillierenden Verhalten beschreibt [LN71, HNW09a, Kor07], u und v bezeichnen die Konzentrationen dieser Substanzen. Das Brusselator-Problem kann durch eine zweidimensionale partielle Differentialgleichung der Form:

$$\frac{\partial u}{\partial t} = 1 + u^2 v - 4.4u + \alpha(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2})$$

$$\frac{\partial v}{\partial t} = 3.4u - u^2 v + \alpha(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2})$$

auf dem Gebiet

$$0 \leq x \leq 1, \quad 0 \leq y \leq 1$$

für Zeit $t \geq 0$, mit den Neumann-Randbedingungen

$$\frac{\partial u}{\partial \mathbf{n}} = 0, \quad \frac{\partial v}{\partial \mathbf{n}} = 0$$

und den Anfangsbedingungen

$$u(x, y, 0) = 0.5 + y, \quad u(x, y, 0) = 0.5 + x$$

modelliert werden. Setzt man für $i, j = 1, \dots, N$

$$x_i = \frac{i-1}{N-1}, \quad y_j = \frac{j-1}{N-1},$$

und definiert

$$U_{ij}(t) = u(x_i, y_j, t), \quad V_{ij}(t) = v(x_i, y_j, t),$$

so erhält man nach erfolgter Diskretisierung mittels der Linienmethode über ein $N \times N$ Gitter mit der Gittergröße $1/(N-1)$ das System aus gewöhnlichen Differentialgleichungen der Dimension $n = 2N^2$ [HNW09a]:

$$U'_{ij} = 1 + U_{ij}^2 V_{ij} - 4.4U_{ij} + \alpha(N-1)^2 \big(U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1} - 4U_{ij}\big)$$

$$V'_{ij} = 3.4U_{ij} - U_{ij}^2 V_{ij} + \alpha(N-1)^2\big(V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1} - 4V_{ij}\big)$$

Berechnet man für $i, j = 1, \ldots, N$ die Werte U_{ij} und V_{ij} abwechselnd in der Reihenfolge

$$U_{11}, V_{11}, U_{12}, V_{12}, \ldots, U_{ij}, V_{ij}, \ldots, U_{NN}, V_{NN},$$

so hat das BRUSS2D-Problem die Zugriffsdistanz von $2N$. Bei BRUSS2D wird für die Funktionsauswertung einer Komponente auf fünf Komponenten des Argumentvektors zugegriffen, die durch einen 5-Punkt-Stern gegeben sind. Nur zur Berechnung der Randpunkte werden weniger als fünf Komponenten gebraucht. Die Laufzeiten der Funktionsauswertungen sind deshalb für die meisten Komponenten konstant, d. h. unabhängig von der Größe des ODE-Systems. Die Laufzeit für die Auswertung der rechten Seite **f** wächst mit der Anzahl der Gleichungen des ODE-Systems [RR00].

Die Diffusionskonstante α hat Einfluss auf die Steifheit des Systems. Für $\alpha = 2 \cdot 10^3$ hat das AWP ein nicht steifes Verhalten [HNW09a, RR00, Kor07]. Die Steifheit des Differentialgleichungssystems ist aber auch von der Feinheit der Ortsdiskretisierung abhängig. Mit steigenden Werten von N wird das System steif.

A.2. Schwingende Saite (STRING)

Dieses Problem beschreibt eine Saite der Länge L, die unter Spannung zwischen zwei Enden fixiert ist. Die schwingende Saite kann durch die eindimensionale Wellengleichung

$$\frac{\partial^2 u}{\partial t^2} = a^2 \frac{\partial^2 u}{\partial x^2}$$

mit den Anfangsbedigungen

$$u(x,0) = f(x), \quad \frac{\partial u}{\partial t}(x,0) = g(x)$$

und den Randbedingungen

$$u(0,t) = 0, \quad u(L,t) = 0$$

beschrieben werden, die von d'Alembert (1717-1783) aufgestellt wurde [Her05]. Dabei gibt die Konstante $c = \sqrt{\frac{T}{\mu'}}$ die Geschwindigkeit der Welle an, wobei T die Spannung und μ die Masse pro Einheitslänge der Saite sind. Der Werte $u(x,t)$ geben die vertikale Verschiebung der Saite über die Zeit hinweg an, $f(x)$ ist die Ausgangsform der Saite und $g(x)$ ist die anfängliche vertikale Geschwindigkeit. Für die Ableitung der Wellengleichung nimmt man an, dass die vertikale Verschiebung klein ist. Durch die Änderung der Spannung T können unterschiedliche Töne erzeugt werden.

Nach Diskretisierung in den Ortsvariablen erhält man das System aus gewöhnlichen Differentialgleichungen der Dimension $n = 2N$ der Gestalt [HNW09a]:

$$U'_i = K^2 + (U_{i-1} - 2U_i + U_{i+1}).$$

Das STRING-Problem hat die Zugriffsdistanz 3.

A.3. Medizinisches Akzo-Nobel-Problem (MEDAKZO)

Dieses Testproblem beschreibt das Eindringen radioaktiv-markierter Antikörper in ein durch Tumor infiziertes Gewebe [MM08, Kor07]. Das mathematische Modell basiert auf einem Reaktions-Diffusion-Modell aus zwei ein-eindimensionalen Gleichungen der Form:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} - kuv, \quad \frac{\partial u}{\partial t} = -kuv, \tag{A.1}$$

die eine chemische Reaktion $A + B \xrightarrow{k} C$ repräsentieren. In dieser Gleichung ist A das radioaktiv-markierter Antikörper, der mit dem Substrat B, dem Tumor infizierten Gewebe reagiert. In der Gleichung (A.1) geben die Größen u und v die Konzentration der Substanzen A und B wieder. Der Antikörper A ist mobil und das Gewebe B immobil. Das Eindringen des Antikörpers in das Gewebe wird auf dem Streifen $S_T = \{f(x,t) : 0 < x < \infty; 0 < t < T\}$ modelliert. Als Anfangs- und Randbedingungen werden

$$u(x,0) = 0, \quad v(x,0) = v_0$$

verwendet, wobei v_0 konstant ist und

$$u(0,t) = \phi(t) \quad \text{für} \quad 0 < t < T$$

gilt.

Durch die Ortsdiskretisierung mit Hilfe der Linienmethode entsteht das folgende ODE-System:

$$y'(t) = f(t,y), \quad y(0) = g$$

mit

$$y \in \mathbb{R}^{2N}, \quad 0 \leq t \leq 20, \quad g = (0, v_0, 0, v_0, \ldots, 0, v_0)^T.$$

Die Funktion f ist gegeben durch:

$$\begin{aligned} f_{2j-1} &= \alpha_j \cdot \frac{y_{2j+1} - y2j - 3}{\Delta\zeta} + \beta_j \cdot \frac{y_{2j-3} - 2y_{2j-1} + y_{2j+1}}{(\Delta\zeta)^2} - ky_{2j-1}y_{2j} \\ f_{2j} &= -ky_{2j}y_{2j-1} \end{aligned} \tag{A.2}$$

mit

$$\alpha_j = \frac{2(j\Delta\zeta - 1)^3}{c^2}, \quad \beta_j = \frac{(j\Delta\zeta - 1)^4}{c^2}, \quad \Delta\zeta = \frac{1}{N}$$

und

$$j = 1, \ldots, N, \quad y_{-1}(t) = \phi(t), \quad y_{2N+1} = y_{2N-1}.$$

Die Infiltrations-Funktion $\phi(t)$ ist gegeben durch:

$$\phi(t) = \begin{cases} 2 & \text{für} \quad t \in (0,5] \\ 0 & \text{für} \quad t \in (5,T],\, T \geq 5. \end{cases}$$

Für die Konstanten werden die Werte $T = 20$, $c = 4$, $k = 100$ und $v_0 = 1$ angenommen. Die Systemgröße bei MEDAKZO-Problem beträgt $n = 2N$, die Zugriffsdistanz ist 2. Für große Werte von N wird das MEDAKZO-Problem steif [Kor07].

A.4. Spitzen-Katastrophe (CUSP)

Dieses ODE-System ist eine Kombination des Spitzen-Katastrophen (engl. cusp catastrophe) Modells für den Nervenimpuls-Mechanismus von Zeeman [Zee72, HNW09a].

$$-\varepsilon y' = y^3 + ay + b$$

und des Van-der-Pol Oszillators

$$\begin{aligned} \frac{\partial y}{\partial t} &= -\frac{1}{\varepsilon}(y^3 + ay + b) + \sigma \frac{\partial^2 y}{\partial x^2} \\ \frac{\partial a}{\partial t} &= b + 0.07v + \sigma \frac{\partial^2 a}{\partial x^2} \\ \frac{\partial b}{\partial t} &= (1 - a^2)b - a - 0.4y + 0.035v + \sigma \frac{\partial^2 b}{\partial x^2} \end{aligned} \tag{A.3}$$

mit

$$v = \frac{u}{u + 0.1}, \quad u = (y - 0.7)(y - 1.3).$$

Nach der Diskretisierung mittels Linienmethode

$$\sigma = 1/144, \quad D = \sigma N^2 = \frac{N^2}{144}, \quad , v_i = \frac{u_i}{u_i + 0.1} \quad u_i = (y_i - 0.7)(y_i - 1.3),$$

den periodischen Randbedingungen

$$\begin{aligned} y_0 &:= y_N, \quad a_0 := a_N, \quad b_0 := b_N, \\ y_{N+1} &:= y_1, \quad a_{N+1} := a_1, \quad b_{N+1} := b_1 \end{aligned}$$

und Anfangsbedingungen

$$y_i(0) = 0, \quad a_i(0) = -2cos(\frac{2i\pi}{N}), \quad b_i(0) = -2sin(\frac{2i\pi}{N})$$

mit $i = 1, \ldots, N$ entsteht ein eindimensionales, gewöhnliches Differentialgleichungssys-

tem der Größe $n = 3N$

$$
\begin{aligned}
y_i' &= \frac{1}{\varepsilon}(y_i^3 + a_i y_i + b_i) + D(y_{i-1} - 2y_i + y_{i+1}) \\
a_i' &= b_i + 0.07v_i + D(a_{i-1} - 2a_i + a_{i+1}) \\
b_i' &= (1 - a_i^2)b_i - a_i - 0.4y_i + 0.035v_i + D(b_{i-1} - 2b_i + b_{i+1})
\end{aligned} \tag{A.4}
$$

Die Wahl der Konstante ε beeinflußt die Steifheit des CUSP-Problems. Für $\varepsilon = 10^{-4}$ ist dieses ODE-System steif. Das resultiernde ODE-System ist dünnbesetzt, besitzt aber aufgrund periodischer Randbedingungen eine unbeschränkte Zugriffsdistanz.

A.5. N-Körper-Standardproblem für Sternhaufen (STARS)

Bei diesem Problem sind N Punktmassen (z. B. Himmelskörper) gegeben, die sich in einem dreidimensionalen Raum $\mathbb{R}^3$ unter Einfluss eines Kraftfeldes (z. B. Gravitationskraft) bewegen [Kor07]. Jeder Körper hat eine feste Masse m_i, befindet sich zum Zeitpunkt t an einer vorgegebenen Position $\mathbf{r_i}$ im $\mathbb{R}^3$ und bewegt sich zu diesem Zeitpunkt mit einer Geschwindigkeit $\mathbf{v_i}$. Durch die Einwirkung des Gravitationsfeldes erfahren die Himmelskörper eine Beschleunigung (1. Newtonsches Grundgesetz). Für ein Himmelskörper i ergibt sich dann folgende Bewegungsgleichung:

$$
\begin{aligned}
\frac{d\mathbf{r}_i}{dt} &= \mathbf{v}_i, \\
\frac{d\mathbf{v}_i}{dt} &= \mathbf{a}_i = \sum_{j=1;j\neq i}^{N} G\, m_j \,\frac{\mathbf{r}_j - \mathbf{r}_i}{|\mathbf{r}_j - \mathbf{r}_i|^3},
\end{aligned} \tag{A.5}
$$

dabei ist G die Gravitationskonstante, $\mathbf{a}_i$ ist die Beschleunigung des Körpers i zum Zeitpunkt t. Zum Zeitpunkt $t = 0$ gilt:

$$\mathbf{v}_i = 0 \quad \text{für} \quad i = 1, \ldots, N.$$

In dieser Arbeit wird angenommen, dass alle Körper die gleiche Masse m haben, die vom Benutzer vorgegeben werden kann. Zur Darstellung der Position der N Körper sind jeweils drei Koordinaten notwendig. Man erhält so ein ODE-System zweiter Ordnung bestehend aus $3N$ Gleichungen. Nach der Transformation in ein ODE-System erster Ordnung beträgt die Systemgröße $n = 6N$ [Kor07]. Die Zugriffdistanz ist nicht beschränkt.

Bei diesem Problem hängt die Beschleunigung eines Körpers von der Gravitationswirkung aller anderen Körpern ab. Für die Berechnung der Beschleunigung eines Sterns zu einem gegebenen Zeitpunkt t muss die Gravitationswirkung zwischen den $N(N-1)$ Himmelskörperpaaren berechnet werden. Dies bedeutet, dass der Rechenaufwand für steigende Werte von N mit der Ordnung $O(N^2)$ ansteigt.

Anhang B.

Verwendete Rechnersysteme

B.1. Cluster: 32 Knoten, jeder besteht aus 2 AMD Opteron DP 246 Prozessoren

Dieses System besteht aus 32 Knoten. Jeder Knoten besteht jeweils aus zwei Prozessoren vom Typ AMD Opteron DP 246, die mit einer Taktfrequenz von 2.0 GHz ausgestattet sind. Die Cache-Parameter des AMD Opteron DP 246 Prozessors sind in der Tabelle B.1 zusammengefaßt.

Stufe	Typ	Größe	Vorkommen	Assoziativität	Zeilengröße
1	Daten	64 KB	pro Kern	2-fach	64 Byte
1	Instruktion	64 KB	pro Kern	2-fach	64 Byte
2	Unified	1 MB	pro Kern	16-fach	64 Byte

Tabelle B.1.: Cache-Parameter des AMD Opteron DP 246 Prozessors.

B.2. Hydra: 4 Quad-Core Intel Xeon E7330 Prozessoren

Das Rechnersystem Hydra besteht aus vier Quad-Core Intel Xeon E7330 Prozessoren, die mit einer Taktfrequenz von 2.4 GHz laufen. Beim vierkernigen Intel Xeon E7330 Prozessor steht je zwei Kernen, die auf einem Die platziert sind, ein gemeinsames 3 MB L2 Cache zur Verfügung. Die Cache-Parameter des Intel Xeon E7330 Prozessors sind in der Tabelle B.2 dargestellt.

Stufe	Typ	Größe	Vorkommen	Assoziativität	Zeilengröße
1	Daten	32 KB	pro Kern	8-fach	64 Byte
1	Instruktion	32 KB	pro Kern	8-fach	64 Byte
2	Unified	3 MB	pro Die (2 Kerne)	12-fach	64 Byte

Tabelle B.2.: Cache-Parameter des Intel Xeon E7330 Prozessors.

B.3. Hydrus: 4 Quad-Core AMD Opteron 8350 Prozessoren

Dieses Rechnersystem ist mit vier Quad-Core AMD Opteron 8350 Prozessoren ausgestattet, die mit der Taktfrequenz von 2.0 GHz arbeiten. Die Cache-Parameter des AMD Opteron 8350 Prozessors sind in der Tabelle B.3 gegeben. Neben jeweils einem 128 KB fassenden L1-Cache (je 64 KB für Daten und Befehle) und dem 512 KB großen L2-Cache pro Kern verfügt der AMD Opteron 8350 Prozessor zusätzlich über einen *geteilten (engl. shared)* 2 MB großen L3-Cache, auf den alle vier Kerne zugreifen können.

Stufe	Typ	Größe	Vorkommen	Assoziativität	Zeilengröße
1	Daten	64 KB	pro Kern	2-fach	64 Byte
1	Instruktion	64 KB	pro Kern	2-fach	64 Byte
2	Unified	512 KB	pro Kern	16-fach	64 Byte
3	Unified	2 MB	pro 4 Kerne	32-fach	64 Byte

Tabelle B.3.: Cache-Parameter des AMD Opteron 8350 Prozessors.

B.4. Leo: 4 Zwölfkern AMD Opteron 6172 Prozessoren

Dieser Server ist mit 4 Zwölfkern AMD Opteron 6172 Prozessoren ausgestattet. Der AMD Opteron 6172 Prozessor besteht aus zwei Siliziumchips (engl. Die), auf denen jeweils 6 Prozessorkerne in einem CPU-Gehäuse platziert sind. Die Tabelle B.4 zeigt die Cache-Parameter des AMD Opteron 6172 Prozessors. Jedem Prozessor-Kern stehen exklusiv 128 KB L1- sowie 512 KB L2-Cache zur Verfügung. Zusätzlich verfügt der AMD Opteron 6172 Prozessor über einen 12 MB großen L3-Cache, auf den 6 Kerne dynamisch zugreifen können. Der Prozessor läuft mit einer Taktfrequenz von 2.1 GHz.

Stufe	Typ	Größe	Vorkommen	Assoziativität	Zeilengröße
1	Daten	64 KB	pro Kern	2-fach	64 Byte
1	Instruktion	64 KB	pro Kern	2-fach	64 Byte
2	Unified	512 KB	pro Kern	16-fach	64 Byte
3	Unified	6 MB	pro Die (6 Kerne)	48-fach	64 Byte

Tabelle B.4.: Cache-Parameter des AMD Opteron 6172 Prozessors.

B.5. 2-Socket-Nehalem-Server: 2 Quad-Core Intel Xeon E5530 Prozessoren

Dieser ist ein 2-Socket Server auf der Basis der Nehalem-Architektur und ist mit 2 Quad-Core Intel Xeon E5530 Prozessoren ausgestattet. Die Nehalem-Architektur weist

eine dreistufige Cache-Hierarchie auf. Jeder Kern besitzt einen exklusiven L1 und L2-Cache, während L3-Cache von allen Kernen geteilt wird. Die Cache-Parameter des Intel Xeon E5530 Prozessors sind in der Tabelle B.5 gegeben. Alle 4 Prozessorkerne sind auf einem Die untergebracht.

Stufe	Typ	Größe	Vorkommen	Assoziativität	Zeilengröße
1	Daten	32 KB	pro Kern	8-fach	64 Byte
1	Instruktion	32KB	pro Kern	4-fach	64 Byte
2	Unified	256 KB	pro Kern	8-fach	64 Byte
3	Unified	8 MB	pro 4 Kerne	16-fach	64 Byte

Tabelle B.5.: Cache-Parameter des Intel Xeon E5530 Prozessors.

B.6. SGI UltraViolet: 104 Intel Westmere-EX-Prozessoren

Das SGI Ultraviolet-System besteht aus 104 Intel Westmere-EX 10-Kern Prozessoren und stellt 3.2 Gbyte Arbeitsspeicher pro Kern zur Verfügung. Alle 1040 Kerne können mit Hilfe der architekturinternen Interconnect-Technologie SGI NUMAlink auf den gemeinsamen Speicher zugreifen. Dieses System wurde vom LRZ-München zur Verfügung gestellt. Die Cache-Parameter des Westmere-EX Prozessors sind in der Tabelle B.6 gegeben.

Stufe	Typ	Größe	Vorkommen	Assoziativität	Zeilengröße
1	Daten	32 KB	pro Kern	8-fach	64 Byte
1	Instruktion	32KB	pro Kern	4-fach	64 Byte
2	Unified	64 KB	pro Kern	8-fach	64 Byte
3	Unified	30 MB	pro 10 Kerne	24-fach	64 Byte

Tabelle B.6.: Cache-Parameter des Intel Westmere-EX-Prozessors.

Literaturverzeichnis

[ABD+90] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof und D. Sorensen: *LAPACK: A Portable Linear Algebra Library for High-performance Computers*, in *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, S. 2–11, IEEE, Los Alamitos, CA, USA, 1990.

[ADD+09] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek und S. Tomov: *Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects*, *Journal of Physics: Conference Series*, 180(1), 2009.

[ADNT11] E. Agullo, J. Dongarra, R. Nath und S. Tomov: *Fully Empirical Autotuned QR Factorization For Multicore Architectures*, in *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, S. 194–205, Springer-Verlag, Berlin, Heidelberg, 2011.

[Aho08] A.V. Aho: *Compiler: Prinzipien, Techniken und Werkzeuge*, Pearson Studium Informatik, Pearson Education Deutschland, 2008.

[AK02] R. Allen und K. Kennedy: *Optimizing Compilers for Modern Architectures: A Dependence Based Approach*, Morgan Kaufmann, 2002.

[Ale77] R. Alexander: *Diagonally Implicit Runge–Kutta Methods for Stiff O.D.E.'s*, *SIAM Journal on Numerical Analysis*, 14(6):1006–1021, Dez. 1977.

[ALSU06] A. V. Aho, M. S. Lam, R. Sethi und J. D. Ullman: *Compilers: Principles, Techniques and Tools (2Nd Edition)*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[AMDA13] Inc. Advanced Micro Devices (AMD): *BIOS and Kernel Developers Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors*, `http://support.amd.com/TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf`, Jan. 2013, [Online; Abgerufen am 05.02.2015].

[ASKL81] W. Abu-Sufah, D. J. Kuck und D. H. Lawrie: *On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations*, *IEEE Transactions on Computers*, 30(5):341–356, 1981.

[ASW+13] R. E. Angulo, V. Springel, S. D. M. White, A. Jenkins, C. M. Baugh und C. S. Frenk: *Scaling relations for galaxy clusters in the Millennium-XXL simulation*, Sept. 2013.

[ATL09] ATLAS: *INSTALL*, `http://github.com/numpy/vendor/blob/master/src/atlas-3.8.3/INSTALL.txt`, 2009, [Online; Abgerufen am 16.05.2014].

[ATNW11] C. Augonnet, S. Thibault, R. Namyst und P. A. Wacrenier: *StarPU: a unified platform for task scheduling on heterogeneous multicore architectures*, *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[BACD97] J. Bilmes, K. Asanovic, C. W. Chin und J. Demmel: *Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology*, in *11th ACM International Conference on Supercomputing (ICS'97)*, S. 340–347, ACM, 1997.

[BBB04] J. Berland, C. Bogey und C. Bailly: *Optimized explicit schemes: matching and boundary schemes, and 4th-order Runge–Kutta algorithm*, *AIAA Pap*, 2814, 2004.

[Ben96] C. Bendtsen: *Highly stable parallel Runge–Kutta methods*, *Applied Numerical Mathematics: Transactions of IMACS*, 21(1):1–8, Juni 1996.

[BJWE92] F. Bodin, W. Jalby, D. Windheiser und C. Eisenbeis: *A Quantitative Algorithm for Data Locality Optimization*, in *In Code Generation-Concepts, Tools, Techniques*, S. 119–145, Springer-Verlag, 1992.

[BLKD09] A. Buttari, J. Langou, J. Kurzak und J. Dongarra: *A class of parallel tiled linear algebra algorithms for multicore architectures*, *Parallel Computing*, 35(1):38–53, 2009.

[Blu70] L. I. Bluestein: *A Linear Filtering Approach to the Computation of the Discrete Fourier Transform*, *IEEE Transactions on Electroacoustics*, 18:451–455, 1970.

[BLYD12] J. H. Byun, R. Lin, K. A. Yelick und J. Demmel: *Autotuning Sparse Matrix-Vector Multiplication for Multicore*, Techn. Ber. UCB/EECS-2012-215, Electrical Engineering and Computer Sciences, University of California at Berkeley, Nov. 2012.

[BML+13] H. Bae, D. Mustafa, J. W. Lee, Aurangzeb, H. Lin, C. Dave, R. Eigenmann und S. P. Midkiff: *The Cetus Source-to-Source Compiler Infrastructure: Overview and Evaluation*, *International Journal of Parallel Programming*, 41(6):753–767, 2013.

[Boc02] H. G. Bock: *Numerische Mathematik I*, 2002, Vorlesungsskript.

[BPT+11] S. Benkner, S. Pllana, J. L. Träff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney und V. Osipov: *PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems*, *IEEE Micro*, 31(5):28–41, 2011.

[BR03] C. Blum und A. Roli: *Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison*, *ACM Computing Surveys*, 35(3):268–308, Sept. 2003.

[Bre04] G. Brenner: *Numerische Simulation auf Hochleistungsrechnern*, `https://www.tu-clausthal.de/presse/tucontact/2004/Mai/tuc1/29.pdf`, 2004, [Online; Abgerufen am 12.01.2015].

[BU10] U. Brinkschulte und T. Ungerer: *Mikrocontroller und Mikroprozessoren*, Springer-Verlag, 2010.

[Bur95] K. Burrage: *Parallel and sequential methods for ordinary differential equations*, Numerical analysis and scientific computation, Oxford University Press, 1995.

[But64] J. C. Butcher: *On Runge-Kutta processes of high order*, *Journal of the Australian Mathematical Society*, 4:179–194, 5 1964.

[But97] D. R. Butenhof: *Programming with POSIX Threads*, Addison-Wesley, Boston, MA, USA, 1997.

[BWH11] P. Balaprakash, S. M. Wild und P. D. Hovland: *Can search algorithms save large-scale automatic performance tuning?*, *Procedia Computer Science*, 4:2136–2145, 2011.

[BWH13] P. Balaprakash, S. M. Wild und P. D. Hovland: *An Experimental Study of Global and Local Search Algorithms in Empirical Performance Tuning*, in *High Performance Computing for Computational Science - VECPAR 2012, 10th International Conference, Kobe, Japan, Revised Selected Papers*, LNCS, S. 261–269, Springer-Verlag, 2013.

[BWN12] P. Balaprakash, S. M. Wild und B. Norris: *SPAPT: Search Problems in Automatic Performance Tuning*, *Procedia Computer Science*, 9(0):1959–1968, 2012, Proceedings of the International Conference on Computational Science.

[Cas05] J. R. Cash: *Efficient Time Integrators in the Numerical Method of Lines*, *Journal of computational and applied mathematics*, 183(2):259–274, Nov. 2005.

[CC05] L. Cheng und J. B. Carter: *Fast Barriers for Scalable ccNUMA Systems*, in *Proceedings of the 2005 International Conference on Parallel Processing*, S. 241–250, IEEE, 2005.

[CCH05] C. Chen, J. Chame und M. Hall: *Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy*, in *International Symposium on Code Generation and Optimization*, S. 111–122, IEEE, Washington, DC, USA, March 2005.

[CDD+96] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker und R. C. Whaley: *ScaLAPACK: A portable linear algebra library for distributed memory computers. Design issues and performance*, in *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*, Bd. 1041 von *LNCS*, S. 95–106, Springer-Verlag, 1996.

[CFA+07] J. Cavazos, G. Fursin, F. V. Agakov, E. V. Bonilla, M. F. P. O'Boyle und O. Temam: *Rapidly Selecting Good Compiler Optimizations using Performance Counters*, in *Proceedings of the International Symposium on Code Generation and Optimization (CGO'07)*, S. 185–197, IEEE, 2007.

[Che07] C. Chen: *Model-guided empirical optimization for memory hierarchy*, Dissertation, University of Southern California, 2007.

[Chi71] F.H. Chipman: *A-stable Runge-Kutta processes*, *BIT Numerical Mathematics*, 11(4):384–388, 1971.

[Chi01] T. M. Chilimbi: *Efficient representations and abstractions for quantifying and exploiting data reference locality*, in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, Bd. 36 von *PLDI '01*, S. 191–202, ACM, 2001.

[Chr11a] M. M. Christen: *Generating and Auto-Tuning Parallel Stencil Codes*, dissertation, Uni Basel, 2011.

[Chr11b] M. M. Christen: *Generating and auto-tuning parallel stencil codes*, Dissertation, University of Basel, 2011.

[CKLV11] P. W. Coteus, J. U. Knickerbocker, C. H. Lam und Y. A. Vlasov: *Technologies for exascale systems*, *IBM Journal of Research and Development*, 55(5):14, 2011.

[Cla96] P. Clauss: *Counting Solutions to Linear and Nonlinear Constraints Through Ehrhart Polynomials: Applications to Analyze and Transform Scientific Programs*, in *Proceedings of the 10th International Conference on Supercomputing*, S. 278–285, ACM, 1996.

[CM69] E. Cuthill und J. McKee: *Reducing the Bandwidth of Sparse Symmetric Matrices*, in *Proceedings of the 1969 24th National Conference*, S. 157–172, ACM, 1969.

[CM95] S. Coleman und K. S. McKinley: *Tile size selection using cache organization and data layout*, in *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, S. 279–290, ACM, 1995.

[CM99] J. Chame und S. Moon: *A Tile Selection Algorithm for Data Locality and Cache Interference*, in *In 1999 ACM International Conference on Supercomputing*, S. 492–499, ACM Press, 1999.

[Com14] CompuGreen: *The Green500*, `http://www.green500.org/lists/green201411/`, 2014, [Online; Abgerufen am 12.01.2015].

[Cor14] Silicon Graphics International Corp.: *SGI UV: Big Brain for No-Limit Computing*, `http://www.sgi.com/pdfs/4377.pdf`, 2013–2014, [Online; Abgerufen am 21.04.2014].

[Cra12] V. Crastan: *Elektrische Energieversorgung 2: Energiewirtschaft und Klimaschutz Elektrizitatswirtschaft, Liberalisierung Kraftwerktechnik und Alternative Stromversorgung, Chemische Energiespeicherung*, Elektrische Energieversorgung, Springer-Verlag, 2012.

[Cro13] B. Crothers: *End of Moore's Law: It's not just about physics*, `http://www.cnet.com/news/end-of-moores-law-its-not-just-about-physics`, 2013, [Online; Abgerufen am 24.04.2014].

[CS89] K. Chadan und P. C. Sabatier: *Inverse problems in quantum scattering theory*, Texts and Monographs in Physics, Springer-Verlag, New York, second Aufl., 1989.

[CSB11] M. Christen, O. Schenk und H. Burkhart: *PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures*, in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, S. 676–687, IEEE, 2011.

[CSV11] J. W. Choi, A. Singh und R. W. Vuduc: *Model-driven autotuning of sparse matrix-vector multiply on GPUs*, in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, S. 115–126, ACM, 2011.

[CT65] J. M. Cooley und J. W. Tukey: *An Algorithm for the Machine Calculation of Complex Fourier Series*, *Mathematics of Computation*, 19:297–301, 1965.

[CW08] J. Chen und W. Watson: *Software barrier performance on dual quad-core Opterons*, in *International Conference on Networking, Architecture, and Storage*, S. 303–309, IEEE, 2008.

[Dah63] G. G. Dahlquist: *A special stability problem for linear multistep methods*, *BIT Numerical Mathematics*, 3(1):27–43, 1963.

[Dat09] K. Datta: *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*, Dissertation, EECS Department, University of California, Berkeley, Dec 2009.

[DB02] P. Deuflhard und F. Bornemann: *Numerische Mathematik. II: Gewöhnliche Differentialgleichungen*, de Gruyter, 2 Aufl., 2002.

[DBLG11] Y. Dotsenko, S. S. Baghsorkhi, B. Lloyd und N. K. Govindaraju: *Auto-tuning of Fast Fourier Transform on Graphics Processors*, *SIGPLAN Notices*, 46(8):257–266, feb 2011.

[DDG+12] J. Dongarra, T. Dong, M. Gates, A. Haidar, S. Tomov und I. Yamazaki: *MAGMA – a New Generation of Linear Algebra Libraries for GPU and Multicore Architecture*, `http://icl.cs.utk.edu/projectsfiles/magma/pubs/25-MAGMA_1.3_SC12.pdf`, 2012, [Online; Abgerufen am 23.05.2014].

[DEK11] U. Dastgeer, J. Enmyren und C. W. Kessler: *Auto-tuning SkePU: A Multi-backend Skeleton Programming Framework for multi-GPU Systems*, in *Proceedings of the 4th International Workshop on Multicore Software Engineering*, IWMSE '11, S. 25–32, ACM, 2011.

[DR08] W. Dahmen und A. Reusken: *Numerik für Ingenieure und Naturwissenschaftler*, Springer-Lehrbuch, Springer-Verlag, 2008.

[dRVP09] F. de Mesmay, A. Rimmel, Y. Voronenko und M. Püschel: *Bandit-Based Optimization on Graphs with Application to Library Performance Tuning*, in *International Conference on Machine Learning (ICML)*, Bd. 382, S. 729–736, ACM, 2009.

[dVP10] F. de Mesmay, Y. Voronenko und M. Püschel: *Offline Library Adaptation Using Automatically Generated Heuristics*, in *International Parallel and Distributed Processing Symposium (IPDPS)*, S. 1–10, IEEE, Atlanta, Georgia, USA, 2010.

[EES13] L. Euler, F. Engel und L. Schlesinger: *Institutiones Calculi Integralis 2nd Part*, Leonhard Euler, Opera Omnia/Opera mathematica, Birkhäuser Basel, 1913.

[EF78] M. C. Easton und R. Fagin: *Cold-start vs. Warm-start Miss Ratios*, *Communications of the ACM*, 21(10):866–872, Okt. 1978.

[EF10] V. Eijkhout und E. Fuentes: *New Advances in Machine Learning*, Kap. Machine Learning for Multi-stage Selection of Numerical Methods, S. 117–136, INTECH, Febr. 2010.

[EGD+05] A. Epshteyn, M. J. Garzarán, G. DeJong, D. A. Padua, G. Ren, X. Li, K. Yotov und K. Pingali: *Analytic Models and Empirical Search: A Hybrid Approach to Code Optimization*, in *LCPC*, Bd. 4339 von *LNCS*, S. 259–273, Springer-Verlag, 2005.

[Ehl68] B. L. Ehle: *High order A-stable methods for the numerical solution of systems of D.E.'s*, *BIT Numerical Mathematics*, 8(4):276–278, 1968.

[EM12] M. Emmett und M. L. Minion: *Toward an efficient parallel in time method for partial differential equations*, *Communications in Applied Mathematics and Computational Science*, 7(1):105–132, 2012.

[FC03] C. Farhat und M. Chandesris: *Time-decomposed parallel time-integrators: theory and feasibility studies for fluid, structure, and fluid-structure applications*, *Internat. J. Numer. Methods Engrg.*, 58(9):1397–1434, 2003.

[FCA05] B. B. Fraguela, M. G. Carmueja und D. Andrade: *Optimal Tile Size Selection Guided by Analytical Models*, in *Parallel Computing: Current & Future Issues of High-End Computing*, Bd. 33, S. 565–572, Central Institute for Applied Mathematics, Sept. 2005.

[Feh70] E. Fehlberg: *Klassische Runge-Kutta-Formeln vierter und niedrigerer Ordnung mit Schrittweiten-Kontrolle und ihre Anwendung auf Wärmeleitungsprobleme*, *Computing*, 6(1-2):61–71, 1970.

[FJ05] M. Frigo und S. G Johnson: *The design and implementation of FFTW3*, *Proceedings of the IEEE*, 93(2):216–231, 2005.

[FKM+11] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam *et al.*: *Milepost GCC: Machine Learning Enabled Self-tuning Compiler*, *International Journal of Parallel Programming*, 39(3):296–327, Juni 2011.

[FML12] H. Falk, P. Marwedel und P. Lokuciejewski: *Reconciling Compilation and Timing Analysis*, in *Advances in Real-Time Systems*, S. 145–170, Springer-Verlag, 2012.

[FST92a] J. Ferrante, V. Sarkar und W. Thrash: *On estimating and enhancing cache effectiveness*, in *Languages and Compilers for Parallel Computing*, Bd. 589 von *LNCS*, S. 328–343, Springer-Verlag, 1992.

[FST92b] J. Ferrante, V. Sarkar und W. Thrash: *On estimating and enhancing cache effectiveness*, in *Languages and Compilers for Parallel Computing*, Bd. 589 von *LNCS*, S. 328–343, Springer-Verlag, 1992.

[FVP09] B. B. Fraguela, Y. Voronenko und M. Püschel: *Automatic Tuning of Discrete Fourier Transforms Driven by Analytical Modeling*, in *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, S. 271–280, IEEE, 2009.

[FYL06] A. Faraj, X. Yuan und D. Lowenthal: *STAR-MPI: Self Tuned Adaptive Routines for MPI Collective Operations*, in *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, S. 199–208, ACM, 2006.

[Gau06] C. F. Gauß: *Abhandlungen zur Methode der kleinsten Quadrate*, Edition classic, VDM Publishing, 2006.

[Gea86] C. W. Gear: *Potential for parallelism in ordinary differential equations*, in *2. International Conference on Computational Mathematics*, Feb 1986.

[GJ09] L. Grüne und O. Junge: *Gewöhnliche Differentialgleichungen: Eine Einführung aus der Perspektive der dynamischen Systeme*, Bachelorkurs Mathematik, Vieweg & Teubner Verlag, 2009.

[Glö14] M. Glöckler: *Simulation mechatronischer Systeme: Grundlagen und technische Anwendung*, Springer, Springer-Verlag, 2014.

[Gol89] D. E. Goldberg: *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st Aufl., 1989.

[Gre08] R. W. Green: *Vectorization Essentials*, `http://software.intel.com/en-us/articles/vectorization-essential`, 2008, [Online; Abgerufen am 05.03.2014].

[Gus94] K. Gustafsson: *Control-theoretic techniques for stepsize selection in implicit Runge-Kutta methods*, *ACM Transactions on Mathematical Software*, 20(4):496–517, Dez. 1994.

[GX93] C. W. Gear und X. Xuhai: *Parallelism Across Time in ODEs*, *Applied Numerical Mathematics*, 11(1-3):45–68, Jan. 1993.

[Her05] R. L. Herman: *An Introduction to Mathematical Physics via Oscillations*, 2005.

[Her06] M. Hermann: *Numerische Mathematik*, Oldenbourg Wissenschaftsverlag, 2006.

[Heu06] H. Heuser: *Gewöhnliche Differentialgleichungen: Einführung in Lehre und Gebrauch*, Mathematische Leitfäden, Teubner, 2006.

[HIG94] M. Horowitz, T. Indermaur und R. Gonzalez: *Low-power digital design*, in *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, S. 8–11, IEEE, Oct 1994.

[HIS+13] D. Hackenberg, T. Ilsche, R. Schöne, D. Molka, M. Schmidt und W. E. Nagel: *Power measurement techniques on standard compute nodes: A quantitative comparison*, in *ISPASS*, S. 194–204, IEEE, 2013.

[HK04] C. H. Hsu und U. Kremer: *A Quantitative Analysis of Tile Size Selection Algorithms*, *The Journal of Supercomputing*, 27(3):279–294, 2004.

[HNW09a] E. Hairer, S. P. Nørsett und G. Wanner: *Solving Ordinary Differential Equations I: Nonstiff Problems*, Springer-Verlag, 2nd Aufl., Dez. 2009.

[HNW09b] E. Hairer, S. P. Nørsett und G. Wanner: *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, Springer-Verlag, 2nd Aufl., Dez. 2009.

[Hof09] R. Hoffmann: *Effiziente taskbasierte ProgrammausfÃ¼hrung irreguljunrer Applikationen mit adaptiver Lastbalancierung*, Dissertation, University of Bayreuth, 2009.

[Hol92] J. H. Holland: *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*, MIT Press, Cambridge, MA, USA, 1992.

[HW10] G. Hager und G. Wellein: *Introduction to High Performance Computing for Scientists and Engineers*, CRC Press, Boca Raton, FL, USA, 1st Aufl., 2010.

[IN90] A. Iserles und S. P. Nørsett: *On the theory of parallel Runge–Kutta methods*, *IMA Journal of numerical Analysis*, 10(4):463–488, 1990.

[Int14a] Intel: *Frequently Asked Questions for Enhanced Intel SpeedStep Technology on Desktop*, `http://www.intel.com/support/de/mt/mt_win.html`, 2014, [Online; Abgerufen am 15.01.2015].

[Int14b] Intel: *Intel Turbo-Boost-Technologie 2.0*, `http://www.intel.de/content/www/de/de/architecture-and-technology/turbo-boost/turbo-boost-technology.html`, 2014, [Online; Abgerufen am 15.01.2015].

[JTD+12] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer und H. Moritsch: *A multi-objective auto-tuning framework for parallel codes*, in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*, S. 10:1–10:12, IEEE, 2012.

[Jül11] Forschungszentrum Jülich: *Newsletter des Forschungszentrums zum Supercomputing - Nr. 03/2011*, `http://www.fz-juelich.de/SharedDocs/Downloads/PORTAL/DE/publikationen/exascale-newsletter/exascale_nl_03_2011.pdf`, 2011, [Online; Abgerufen am 12.01.2015].

[Kab08] S. Kabanikhin: *Definitions and examples of inverse and ill-posed problems*, *Journal of Inverse and Ill-Posed Problems*, 16(4):317–357, 2008.

[Kas06] J. Kaschenz: *Regularisierung unter Berücksichtigung von Residuentoleranzen*, Dissertation, Technische Universität Berlin, 2006.

[KCL00] C. A. Kennedy, M. H. Carpenter und R. M. Lewis: *Low-storage, explicit Runge–Kutta schemes for the compressible Navier–Stokes equations*, *Applied Numerical Mathematics: Transactions of IMACS*, 35(3):177–219, Nov. 2000.

[KCO+10] S. Kamil, C. Chan, L. Oliker, J. Shalf und S. Williams: *An auto-tuning framework for parallel multicore stencil computations*, in *Proceedings of the International Parallel and Distributed Processing Symposium*, S. 1–12, IEEE, 2010.

[KCW+10] S. Kamil, C. Chan, S. Williams, L. Oliker, J. Shalf, M. Howison und E. W. Bethel: *A Generalized Framework for Auto-tuning Stencil Computations*, in *2010 IEEE International Symposium on Parallel Distributed Processing*, S. 1–12, IEEE, April 2010.

[Ken10] J. Kennedy: *Particle swarm optimization*, in *Encyclopedia of Machine Learning*, S. 760–766, Springer-Verlag, 2010.

[KGCF13] K. Kofler, I. Grasso, B. Cosenza und T. Fahringer: *An Automatic Input-sensitive Approach for Heterogeneous Task Partitioning*, in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, S. 149–160, ACM, 2013.

[KGV83] S. Kirkpatrick, C. D. Gelatt und M. P. Vecchi: *Optimization by simulated annealing*, *SCIENCE*, 220(4598):671–680, 1983.

[KKO00] T. Kisuki, P. M. W. Knijnenburg und M. F. P. O'Boyle: *Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation*, in *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT '00)*, S. 237–248, IEEE, 2000.

[KKR10] N. Kalinnik, M. Korch und T. Rauber: *Applicability of Dynamic Selection of Implementation Variants of Sequential Iterated Runge-Kutta Methods*, in *2010 IEEE International Conference on Cluster Computing – Workshops and Tutorials*, IEEE, 2010.

[KKR11a] N. Kalinnik, M. Korch und T. Rauber: *Dynamic Selection of Implementation Variants of Sequential Iterated Runge-Kutta Methods with Tile Size Sampling*, in *Proceedings of the Second Joint WOSP/SIPEW International Conference on Performance Engineering*, S. 189–200, ACM, 2011.

[KKR11b] N. Kalinnik, M. Korch und T. Rauber: *An efficient time-step-based self-adaptive algorithm for predictor-corrector methods of Runge-Kutta type*, *J. Computational Applied Mathematics*, 236(3):394–410, 2011.

[KKR14] N. Kalinnik, M. Korch und T. Rauber: *Online auto-tuning for the time-step-based parallel solution of ODEs on shared-memory systems*, *J. Parallel Distrib. Computing*, 74(8):2722–2744, 2014.

[Kor07] M. Korch: *Effiziente Implementierung eingebetteter Runge-Kutta-Verfahren durch Ausnutzung der Speicherzugriffslokalität*, Dissertation, University of Bayreuth, 2007.

[Kor12] M. Korch: *Exploiting Limited Access Distance of ODE Systems for Parallelism and Locality in Explicit Methods*, in *ALGORITMY 2012. 19th Conference on Scientific Coputing, Vysoké Tatry – Podbanské, Slovakia, September 9–14, 2012. Proceedings of contributed papers and posters*, S. 250–260, Slovak University of Technology in Bratislava, Faculty of Civil Engineering, Department of Mathematics and Descriptive Geometry, 2012.

[KP11a] T. Karcher und V. Pankratius: *Auto-Tuning Multicore Applications at Run-Time with a Cooperative Tuner*, Karlsruhe Reports in Informatics 2011,4, Karlsruhe Institute of Technology, Faculty of Informatics, Febr. 2011.

[KP11b] T. Karcher und V. Pankratius: *Run-Time Automatic Performance Tuning for Multicore Applications*, in *Euro-Par 2011. Part I*, Nr. 6852 in LNCS, S. 3–14, Springer-Verlag, 2011.

[KR06] M. Korch und T. Rauber: *Optimizing locality and scalability of embedded Runge-Kutta solvers using block-based pipelining*, *J. Parallel Distrib. Comput*, 66(3):444–468, 2006.

[KR07a] M. Korch und T. Rauber: *Locality Optimized Shared-Memory Implementations of Iterated Runge-Kutta Methods*, Presentation at the 13th International Euro-Par Conference, 2007.

[KR07b] M. Korch und T. Rauber: *Locality Optimized Shared-Memory Implementations of Iterated Runge-Kutta Methods*, in *Parallel Processing, 13th International Euro-Par Conference*, Bd. 4641, S. 737–747, Springer-Verlag, 2007.

[KR11] M. Korch und T. Rauber: *Parallel Low-Storage Runge - Kutta Solvers for ODE Systems with Limited Access Distance*, *IJHPCA*, 25(2):236–255, 2011.

[KW02] M. Kowarschik und C. Weiß: *An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms*, in *Algorithms for Memory Hierarchies*, Bd. 2625 von *LNCS*, S. 213–232, Springer, 2002.

[LA04] C. Lattner und V. Adve: *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*, in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, S. 75–88, IEEE, Palo Alto, California, Mar 2004.

[Lan14] J. Lang: *Grüner verschlüsseln – Messung des Energieverbrauchs von Verschlüsselungsalgorithmen*, in *Chemnitzer Linux-Tage 2014 – Tagungsband*, S. 25–32, Universitätsverlag Chemnitz, 2014.

[LDT09] Y. Li, J. Dongarra und S. Tomov: *A Note on Auto-tuning GEMM for GPUs*, in *Computational Science – (9th ICCS'09, Part I)*, Bd. 5544 von *LNCS*, S. 884–892, Springer-Verlag, Baton Rouge, LA, USA, Mai 2009.

[LMT01] J. L. Lions, Y. M. und G. Turinici: *A parareal in time discretization of PDE's*, *C. R. Acad. Sci. Paris, Serie I*, 332:661–668, 2001.

[LN71] R. Lefever und G. Nicolis: *Chemical instabilities and sustained oscillations*, *Journal of Theoretical Biology*, 30(2):267–284, 1971.

[LRW91] M. S. Lam, E. E. Rothberg und M. E. Wolf: *The cache performance and optimizations of block algorithms*, in *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, S. 63–74, ACM, 1991.

[Luk05] Stefan Lukowitz: *Ermittlung des Quellcodebezugs von Speicherzugriffen auf Datenobjekte*, Lehrstuhl für Rechnertechnik und Rechnerorganisation/Parallelrechnerarchitektur (LRR), Informatik 10, Fakultät für Informatik, Technische Universität München, Germany, Mai 2005, Diplomarbeit.

[Man10] P. Mandl: *Grundkurs Betriebssysteme*, Springer-Studium, Springer-Verlag, 2010.

[Man14] K. Manhart: *Grenzen der Prozessor-Technik erreicht*, `http://www.pc-magazin.de/ratgeber/moore-law-report-ende-2020-1938131.html`, 2014, [Online; Abgerufen am 24.04.2014].

[Mar01] A. J. Martin: *Towards an energy complexity of computation*, *Information Processing Letters*, 77(2-4):181–187, 2001.

[MBY13] S. Mehta, G. Beeraka und P. C. Yew: *Tile Size Selection Revisited*, *CM Transactions on Architecture and Code Optimization*, 10(4):1–27, Dez. 2013.

[MC69] A. C. McKellar und E. G. Coffman, Jr.: *Organizing Matrices and Matrix Operations for Paged Memory Systems*, *Communications of the ACM*, 12(3):153–165, März 1969.

[MCP08] R. Martí, V. Campos und E. Piñana: *A branch and bound algorithm for the matrix bandwidth minimization*, *European Journal of Operational Research*, 186(2):513–528, 2008.

[Mer13] A. Mertins: *Signaltheorie: Grundlagen Der Signalbeschreibung, Filterbänke, Wavelets, Zeit-Frequenz-Analyse, Parameter- und Signalschätzung*, SpringerLink : Bücher, Springer-Verlag, 2013.

[MJ01] D. Mirkoviѷ und S. L. Johnsson: *Automatic Performance Tuning in the UHFFT Library*, in *Computational Science*, Bd. 2073 von *LNCS*, S. 71–80, Springer-Verlag Berlin Heidelberg, 2001.

[MM08] F. Mazzia und C. Magherini: *Test set for initial value problem solvers, release 2.4*, Technical Report 4, Department of Mathematics, University of Bari, Italy, February 2008.

[Moo65] G. E. Moore: *Cramming more components onto integrated circuits*, *Electronics*, 38(8), April 1965.

[MSS03] U. Meyer, P. Sanders und J. Sibeyn (Hg.): *Algorithms for Memory Hierarchies: Advanced Lectures*, Springer-Verlag, Berlin, Heidelberg, 2003.

[Muc97] S. S. Muchnick: *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[MV06] K. Meyberg und P. Vachenauer: *Höhere Mathematik 2: Differentialgleichungen, Funktionentheorie, Fourier-Analysis, Variationsrechnung*, Höhere Mathematik, Springer-Verlag, 2006.

[NM65] J. A. Nelder und R. Mead: *A simplex method for function minimization*, *Computer Journal*, 7:308–313, 1965.

[NS89] S. P. Nørsett und H. H. Simonsen: *Aspects of parallel Runge-Kutta methods*, in *Numerical Methods for Ordinary Differential Equations*, S. 103–117, Springer-Verlag, 1989.

[OKJ+13] T. Oh, H. Kim, N. P. Johnson, J. W. Lee und D. I. August: *Practical automatic loop specialization*, in *Architectural Support for Programming Languages and Operating Systems, ASPLOS'13*, S. 419–430, ACM, 2013.

[ope14] *The OpenMP API specification for parallel programming*, `http://openmp.org/wp`, last checked: 10/2014.

[Pap76] C. H. Papadimitriou: *The NP-completeness of the bandwidth minimization problem*, *Computing*, 16:263–270, 1976.

[pap14] *Performance Application Programming Interface (PAPI) Homepage*, `http://icl.cs.utk.edu/papi/`, last checked: 07/2014.

[PD81] P. J. Prince und J. R. Dormand: *High order embedded Runge-Kutta formulae*, *Journal of Computational and Applied Mathematics*, 7(1):67–75, 1981.

[PE08] Z. Pan und R. Eigenmann: *PEAK&Mdash;a Fast and Effective Performance Tuning System via Compiler Optimization Orchestration*, *ACM Trans. Program. Lang. Syst.*, 30(3):17:1–17:43, Mai 2008.

[PFV11] M. Püschel, F. Franchetti und Y. Voronenko: *Encyclopedia of Parallel Computing*, Kap. Spiral, Springer-Verlag, 2011.

[PH11] D. Patterson und J.L.R. Hennessy: *Rechnerorganisation und Rechnerentwurf: Die Hardware/Software-Schnittstelle*, Oldenbourg Wissenschaftsverlag, 2011.

[PH13] D.A. Patterson und J.L. Hennessy: *Computer Organization and Design: The Hardware/Software Interface*, The Morgan Kaufmann Series in Computer Architecture and Design, Elsevier Science, 2013.

[PKHW04] R. P. J. Pinkers, P. M. W. Knijnenburg, M. Haneda und H. A. G. Wijshoff: *Statistical selection of compiler options*, in *Proceedings of the IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, MASCOTS '04, S. 494–501, IEEE, Washington, DC, USA, 2004.

[Pla10] R. Plato: *Numerische Mathematik kompakt: Grundlagenwissen für Studium und Praxis*, Numerische Mathematik, Vieweg, 2010.

[PM02] P. I. Pénzes und A. J. Martin: *Energy-delay efficiency of VLSI computations*, in *Proceedings of the 12th ACM Great Lakes Symposium on VLSI*, S. 104–111, ACM, 2002.

[PMJ+05] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson und N. Rizzolo: *SPIRAL: Code Generation for DSP Transforms*, *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.

[PMS+04] M. PÃ¼schel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso und R. W. Johnson: *SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms*, *Journal of High Performance Computing and Applications*, 18:21–45, 2004.

[PnPCM04] E. Piñana, I. Plana, V. Campos und R. Martí: *GRASP and path relinking for the matrix bandwidth minimization*, *European Journal of Operational Research*, 153(1):200–210, 2004.

[QL11] D. Quinlan und C. Liao: *The ROSE Source-to-Source Compiler Infrastructure*, in *Cetus Users and Compiler Infrastructure Workshop*, S. 1, Okt. 2011.

[QM12] J. Quade und M. Mächtel: *Moderne Realzeitsysteme kompakt*, Dpunkt-Verlag, Heidelberg, 2012.

[Ris13] V. Risska: *Intel: 50-fache Performance in sechs Jahren im HPC-Segment*, `http://www.computerbase.de/2013-10/intel-50-fache-performance-in-sechs-jahren-im-hpc-segment`, 2013, [Online; Abgerufen am 14.04.2014].

[RNA+12] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann und D. Rajwan: *Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge*, *IEEE Micro*, 32(2):20–27, 2012.

[RR95] T. Rauber und G. Rünger: *Performance Predictions for Parallel Diagonal-Implicitly Iterated Runge–Kutta Methods*, in *9th Workshop on Parallel and Distributed Simulation*, S. 21–28, 1995.

[RR96] T. Rauber und G. Rünger: *Parallel Implementations of Iterated Runge-Kutta Methods*, *International Journal of Supercomputer Applications and High Performance Computing*, 10(1):62–90, 1996.

[RR99] T. Rauber und G. Rünger: *Diagonal-Implicitly Iterated Runge–Kutta Methods on Distributed Memory Machines*, *International Journal of High Speed Computing*, 10(2):185–207, 1999.

[RR00] T. Rauber und G. Rünger: *Parallele Und Verteilte Programmierung*, Springer-Lehrbuch, Springer-Verlag, 2000.

[RR04] T. Rauber und G. Rünger: *Improving locality for ODE solvers by program transformations*, *Scientific Programming*, 12(3):133–154, 2004.

[RR12a] T. Rauber und G. Rünger: *Parallele Programmierung*, Springer-Verlag Berlin, 2012.

[RR12b] T. Rauber und G. Rünger: *Parallele Programmierung*, Springer-Verlag London, 2012.

[RS13] L. Rios und N. Sahinidis: *Derivative-free optimization: a review of algorithms and comparison of software implementations*, *Journal of Global Optimization*, 56(3):1247–1293, 2013.

[RT97] G. Rivera und C.-W. Tseng: *Compiler Optimizations for Eliminating Cache Conflict Misses*, Technical Report CS-TR-3819, University of Maryland, College Park, Juli 1997.

[RU08] L. Renganarayana und Colorado State University: *Scalable and Efficient Tools for Multi-level Tiling*, Colorado State University, 2008.

[Ruu05] S. J. Ruuth: *Global optimization of explicit strong-stability-preserving Runge–Kutta methods*, *Mathematics of Computation*, 75(253):183–207, 2005.

[Sab12] M. Sabahi: *A Guide to Vectorization with Intel C++ Compilers*, `http://download-software.intel.com/sites/default/files/8c/a9/CompilerAutovectorizationGuide.pdf`, 2012, [Online; Abgerufen am 20.03.2014].

[SK11] H. R. Schwarz und N. Köckler: *Numerische Mathematik*, Vieweg+Teubner Verlag, 2011.

[Smi00] M. D. Smith: *Overcoming the Challenges to Feedback-directed Optimization (Keynote Talk)*, in *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, DYNAMO '00, S. 1–11, ACM, 2000.

[Soe06] G. Soederlind: *Time-step selection algorithms: adaptivity, control, and signal processing*, *Applied Numerical Mathematics*, 56(3-4):488–502, März 2006.

[SS07] Y. N. Srikant und P. Shankar: *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*, Taylor & Francis, 2007.

[SSF+12] J. Shirako, K. Sharma, N. Fauzia, L. N. Pouchet, J. Ramanujam, P. Sadayappan und V. Sarkar: *Analytical Bounds for Optimal Tile Size Selection*, in *ETAPS International Conference on Compiler Construction (CC'12)*, S. 101–121, Springer-Verlag, 2012.

[STD12] F. Song, S. Tomov und J. Dongarra: *Enabling and Scaling Matrix Computations on Heterogeneous Multi-core and multi-GPU Systems*, in *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, S. 365–376, ACM, 2012.

[Str12] M. Straubinger: *Reduktion des Kommunikationsaufwands iterierter Runge-Kutta-Verfahren für dünnbesetzte gewöhnliche Differentialgleichungssysteme*, Bachelorarbeit, Universität Bayreuth, Deutschland, 2012.

[Sue03] D. Suess: *Computational Physics, Ausgewählte Kapitel des Skriptums*, `http://magnet.atp.tuwien.ac.at/suess/cp/cp/linear.pdf`, 2003, [Online; Abgerufen am 02.04.2014].

[SWP12] K. Strehmel, R. Weiner und H. Podhaisky: *Numerik gewöhnlicher Differentialgleichungen: Nichtsteife, steife und differential-algebraische Gleichungen*, Springer-Link : Bücher, Vieweg & Teubner Verlag, 2012.

[SYD08] K. Seymour, H. You und J. Dongarra: *A comparison of search heuristics for empirical code optimization*, in *CLUSTER*, S. 421–429, IEEE, 2008.

[Tan09] A.S. Tanenbaum: *Moderne Betriebssysteme*, Pearson Studium - IT, Pearson Deutschland, 2009.

[TH11] A. Tiwari und J. K. Hollingsworth: *Online Adaptive Code Generation and Tuning*, in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, S. 879–892, IEEE, 2011.

[THW02] H. Topcuoglu, S. Hariri und M.-Y. Wu: *Performance-effective and low-complexity task scheduling for heterogeneous computing*, *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, Mar 2002.

[Tik95] A. N. Tikhonov: *Numerical methods for the solution of ill-posed problems*, Bd. 328, Kluwer Academic Publishers, 1995.

[Tod50] J. Todd: *Solution of Differential Equations by Recurrence Relations*, *Mathematical Tables and Other Aids to Computation*, 4(29):39–44, 1950.

[top14] top500: *November, 2014*, `http://top500.org/lists/2014/11/`, 2014, [Online; Abgerufen am 12.01.2015].

[Ung10] T. Ungerer: *Mikrocontroller und Mikroprozessoren*, Springer-Verlag, 2010.

[vdHS90a] P. J. van der Houwen und B. P. Sommeijer: *Parallel iteration of high-order Runge–Kutta Methods with stepsize control*, *Journal of Computational and Applied Mathematics*, 29:111–127, 1990.

[vdHS90b] P. J. van der Houwen und B. P. Sommeijer: *Parallel ODE solvers*, in *Proceedings of the 4th international conference on Supercomputing*, ICS'90, S. 71–81, ACM, 1990.

[vdHSC92] P. J. van der Houwen, B. P. Sommeijer und W. Couzy: *Embedded diagonally implicit Runge–Kutta algorithms on parallel computers*, *Mathematics of Computation*, 58(197):135–159, jan 1992.

[vdHSvdV94] P. J. van der Houwen, B. P. Sommeijer und W. A. van der Veen: *Parallelism across the steps in iterated Runge-Kutta methods for stiff initial value problems*, *Numer. Algorithms*, 8(2-4):293–312, 1994.

[vdHSvdV95] P. J. van der Houwen, B. P. Sommeijer und W. A. van der Veen: *Parallel iteration across the steps of high-order Runge-Kutta methods for nonstiff initial value problems*, *J. Comput. Appl. Math.*, 60(3):309–329, 1995.

[VDY05] R. Vuduc, J. W. Demmel und K. A. Yelick: *OSKI: A library of automatically tuned sparse matrix kernels*, *Journal of Physics: Conference Series. Proceedings of SciDAC 2005*, 16:521–530, 2005.

[Völ11] Lars Völker: *Untersuchung des Kommunikationsintervalls bei der gekoppelten Simulation*, Bd. 6, KIT Scientific Publishing, 2011.

[Vud11] R. W. Vuduc: *Autotuning*, in *Encyclopedia of Parallel Computing*, Bd. 4, S. 102–105, Springer Science & Business Media, 2011.

[WD97] R. C. Whaley und J. J. Dongarra: *Automatically Tuned Linear Algebra Software*, Techn. Ber. UT-CS-97-366, University of Tennessee, 1997.

[Wei14] F. Wein: *Auto-Tuning iterierter Runge-Kutta-Verfahren unter Berücksichtigung des Energieverbrauchs*, Masterarbeit, Universität Bayreuth, Deutschland, 2014.

[WGS09] Q. Wang, Y. C. Guo und X. W. Shi: *An improved algorithm for matrix bandwidth and profile reduction in finite element analysis*, *Progress In Electromagnetics Research Letters*, 9:29–38, 2009.

[Wik13] Wikipedia: *Intrinsische Funktion*, `http://de.wikipedia.org/wiki/Intrinsische_Funktion`, 2013, [Online; Abgerufen am 18.02.2014].

[Wol89] M. Wolfe: *Iteration Space Tiling for Memory Hierarchies*, in *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, S. 357–361, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1989.

[Wol95] M. J. Wolfe: *High Performance Compilers for Parallel Computing*, Addison-Wesley Longman Publishing Co., Boston, MA, USA, 1995.

[WPD01] R. C. Whaley, A. Petitet und J. J. Dongarra: *Automated Empirical Optimization of Software and the ATLAS Project*, *Parallel Computing*, 27(1–2):3–25, 2001.

[XJJP01] Jianxin X., J. Johnson, R. Johnson und D. Padua: *SPL: A Language and Compiler for DSP Algorithms*, in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, S. 298–308, ACM, 2001.

[YLR+05] K. Yotov, X. Li, G. Ren, M. J. S. Garzaran, D. Padua, K. Pingali und P. Stodghill: *Is Search Really Necessary to Generate High-Performance BLAS?*, *Proceedings of the IEEE*, 93(2):358–386, 2005.

[Zee72] E. C. Zeeman: *Differential equations for the heartbeat and nerve impulse*, *Towards a theoretical biology*, 4:8–67, 1972.