View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by EPub Bayreuth

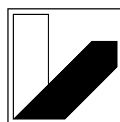
Effiziente taskbasierte Programmausführung irregulärer Applikationen mit adaptiver Lastbalancierung

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

von

Ralf Hoffmann

aus Halle (Saale)


UNIVERSITÄT
BAYREUTH

1.Gutachter: Prof. Dr. Thomas Rauber
2.Gutachter: Prof. Dr. Jörg Keller

Tag der Einreichung: 2.Februar 2009
Tag des Kolloquiums: 4.Mai 2009

Effiziente taskbasierte Programmausführung irregulärer Applikationen mit adaptiver Lastbalancierung

Kurzfassung

Zur parallelen Ausführung irregulärer Applikationen auf Parallelrechnern mit gemeinsamem Speicher eignet sich ein taskbasierter Ansatz, da durch eine dynamische Lastverteilung einzelner Tasks geeignet auf die irreguläre Abarbeitungsstruktur der Applikation reagiert werden kann. In dieser Arbeit wird das KOALA-Framework zur Abarbeitung feingranularer Tasks paralleler Programme vorgestellt. Es wird ein adaptiver Taskpool beschrieben, der effizient auf wechselnde Lastzustände innerhalb einer Applikation reagieren kann. Durch adaptiv angepasste Taskblöcke kann der adaptive Taskpool auch bei einer sehr großen Anzahl von ausführungsbereiten Tasks mit nur wenigen Operationen Taskumverteilungen durchführen, um die Berechnungen möglichst gleichmäßig auf die Prozessoren zu verteilen. Verschiedene irreguläre Applikationen werden getestet, um die unterschiedlichen Taskpool-Implementierungen zu vergleichen. Dabei erzielen die adaptiven Taskpools im Gegensatz zu konventionellen Taskpools bei allen untersuchten Applikationen gute Ergebnisse.

Die einzelnen funktionalen Bestandteile des KOALA-Frameworks können ohne Änderungen in der Applikation ausgetauscht werden. Eine spezielle Implementierung der Lock-Komponente erlaubt so die Nutzung von Hardware-Operationen zur effizienten Synchronisation der beteiligten Threads. Weiterhin wird eine Profiling-Komponente vorgestellt, mit der die Taskstruktur einer Applikation analysiert werden kann. An einem Fallbeispiel werden Engstellen in einer Applikation identifiziert, durch deren Behebung eine erhebliche Verbesserung der Laufzeit erreicht werden konnte.

Efficient Task-Based Execution of Irregular Applications with Adaptive Load Balancing

Abstract

This thesis considers the execution of irregular applications on shared-memory machines with many processor cores. A task-based approach with dynamic load balancing can be used to handle the irregular execution scheme of such applications. The KOALA framework presented in this thesis is used to execute fine-grained tasks of arbitrary task-based applications. An adaptive task pool is developed which is able to handle a large amount of tasks efficiently. By using dynamically adjusted blocks of tasks the adaptive task pool is able to balance the work between processors almost independently from the actual number of tasks involved. The task pool implementations are compared using several irregular applications. In contrast to conventional task pool implementations, the adaptive task pools have shown good performance for all applications.

The KOALA framework is divided into several components to enable the use of different implementations. A specific implementation of the lock component uses hardware operations for efficient synchronization. Additionally, a profiling component is presented which can be used to analyze the task structure of a given application. Bottlenecks of task-based applications are identified in a case study. After improving the performance critical parts of the application, a significant runtime improvement could be achieved.

Danksagung

Ganz besonders danken möchte ich Prof. Dr. Thomas Rauber für die Möglichkeit, in dem interessanten Forschungsgebiet der Parallelverarbeitung zu promovieren, und für seine Unterstützung bei den erfolgten Veröffentlichungen. Mein Dank gilt außerdem meinen Kollegen des Lehrstuhls der Angewandten Informatik II der Universität Bayreuth. Ohne den Erfahrungsaustausch hätten die aufgetauchten Probleme sicher nicht so schnell gelöst werden können und so manche Idee wäre gar nicht erst entstanden. Besonderer Dank gilt Dr. Sascha Hunold, Dr. Matthias Korch und Raik Nagel für die gute Zusammenarbeit zur Lösung alltäglicher Probleme im Forschungsbetrieb und für konstruktive Kritik, die zum Gelingen dieser Arbeit beitrug.

Natürlich möchte ich auch meinen Eltern danken, die mich immer unterstützt haben und mir den nötigen Freiraum gegeben haben, die gesteckten Ziele zu erreichen.

Inhaltsverzeichnis

1 Einleitung **17**
1.1 Ziele und Gliederung . . . 18
1.2 Begriffsklärung . . . 19
1.2.1 Zielarchitekturen . . . 19
1.2.2 Parallele Ausführungsumgebung . . . 22
1.2.3 Lastbalancierung . . . 24
1.2.4 Leistungsbewertung . . . 26
1.2.5 Betrachtete Applikationsklasse . . . 27

2 Das Taskpool-Programmiermodell und das KOALA-Framework **29**
2.1 Parallele Programmausführung mit Tasks . . . 29
2.2 Aufbau des KOALA-Frameworks . . . 33
2.3 Speichermanager . . . 36
2.3.1 Schnittstellen für die Applikation . . . 36
2.3.2 Schnittstellen für die Taskpool-Schicht . . . 37
2.4 Lock-Komponente . . . 38
2.4.1 Lock-Interface . . . 38
2.4.2 Duolock-Interface . . . 39
2.4.3 Barrier-Interface . . . 40
2.5 Unterstützende Tools . . . 42
2.6 Backend . . . 44
2.6.1 Interface . . . 44
2.7 Frontend . . . 45
2.7.1 Interface . . . 45
2.7.2 Aufgaben . . . 46
2.8 Beispielapplikation . . . 47

3 Taskbasierte Applikationen **49**
3.1 Synthetische Applikation . . . 49
3.2 Ray-Tracing . . . 51
3.3 Volume-Rendering . . . 52
3.4 Hierarchisches Radiosity . . . 53
3.5 Quicksort . . . 56

4 Performance-Analyse Hardware-unterstützter Taskpools **59**
4.1 Einführung . . . 59
4.2 Zielarchitekturen und Synchronisationsoperationen . . . 61
4.3 Implementierung der Taskpools . . . 64

4.3.1 Speichermanager 64
4.3.2 Zentrale Taskpools 65
4.3.3 Verteilte Taskpools ohne Taskgruppierung 74
4.3.4 Blockverteilte Taskpools 80
4.4 Laufzeitergebnisse mit realistischen Applikationen 84
4.4.1 Hierarchisches Radiosity 84
4.4.2 Paralleles Quicksort 86
4.4.3 Volume-Rendering 88
4.4.4 Ray-Tracing 89
4.5 Zusammenfassung 90

5 Implementierung des KOALA-Frameworks 93
5.1 Speichermanager 93
5.1.1 Speichermanager „membs“ 93
5.1.2 Blockbasierter Speichermanager „memdgbk“ 94
5.1.3 Statischer Speichermanager „memstatic“ 94
5.1.4 Zugriffsoptimierter Speichermanager „memopt“ 95
5.2 Lock-Komponente 98
5.2.1 Pthreads-Lock „mutexlock“ 98
5.2.2 Pthreads-Spinlock „spinlock“ 99
5.2.3 Hardware-Lock „hwlock“ 100
5.2.4 Hardware-Ticket-Lock „hwticketlock“ 103
5.2.5 Duolock 106
5.3 Unterstützende Funktionen 109
5.3.1 Code- und Speicherbarrieren 109
5.3.2 Parallele Speicherverwaltung „memlist“ 110
5.3.3 Zeitmessung mit Hardware-Unterstützung „hw_timesnap“ 112
5.4 Automatisches Tuning Hardware-abhängiger Parameter 117
5.5 Taskpool-Backend-Implementierungen 120
5.5.1 Grundlagen 121
5.5.2 Zentrale Taskpools 121
5.5.3 Verteilte Taskpools 121
5.5.4 Blockverteilte Taskpools 122
5.6 Frontend-Implementierungen 123
5.6.1 Frontend-Implementierung „level0“ ohne Profiling 123

6 Adaptive Lastbalancierung 127
6.1 Adaptive Datenstruktur 128
6.1.1 Einfügen neuer Tasks 129
6.1.2 Entnahme von Tasks 130
6.1.3 Task-Stealing 132
6.1.4 Komplexitätsbetrachtungen 133
6.2 Adaptive Taskpool-Implementierung 135
6.3 Experimentelle Ergebnisse 138
6.3.1 Synthetische Applikation 138

6.3.2 Quicksort . . . 140
6.3.3 Ray-Tracing . . . 140
6.3.4 Hierarchisches Radiosity . . . 141
6.4 Zusammenfassung . . . 142

7 Analyse taskbasierter Applikationen mittels Task-Profiling 143
7.1 Profiling-Komponente im Frontend . . . 145
7.1.1 Begriffsklärung . . . 145
7.1.2 Implementierung der Komponente . . . 146
7.1.3 Logarithmisches Histogramm . . . 147
7.2 Auswertung der Profiling-Informationen . . . 149
7.3 Fallbeispiele für die Applikationsanalyse . . . 153
7.3.1 Untersuchung der Radiosity-Applikation . . . 153
7.3.2 Ray-Tracing . . . 156
7.3.3 Volume-Rendering . . . 157
7.3.4 Quicksort . . . 158
7.4 Skalierbarkeitsvorhersage . . . 159
7.4.1 Methoden der Extrapolation . . . 160
7.4.2 Extrapolation am Beispiel der Radiosity-Applikation . . . 167
7.5 Identifizierung von Cache-Fehlzugriffen aufgrund von False-Sharing . . . 169
7.5.1 Messung der False-Sharing-Effekte . . . 170
7.5.2 Analyse-Tool „Valgrind“ . . . 170
7.5.3 Cachetool-Plugin . . . 171
7.5.4 Analysetool . . . 172
7.6 Verbesserung der Radiosity-Applikation . . . 172
7.7 Zusammenfassung . . . 175

8 Zusammenfassung und Ausblick 177

A Rechnersysteme 181
A.1 IBM eServer pSeries 690 . . . 181
A.2 SGI Altix 4700 . . . 181
A.3 Sun Fire 6800 . . . 182

B Zugehörige Publikationen 183

Literaturverzeichnis 185

Abbildungsverzeichnis

2.1 Schema einer parallelen Programmausführung mit einem Taskpool 32
2.2 Schematischer Aufbau des KOALA-Frameworks 34
2.3 Schnittstelle der Operationen für wechselseitigen Ausschluss 38
2.4 Schnittstelle der zweistufigen Lock-Operationen 39
2.5 Schnittstelle der Barrier-Operationen 40
2.6 Schnittstelle der Zeitmessungen 42
2.7 Schnittstelle der Hardware-unterstützten Zeitmessungen 43
2.8 Schnittstelle der parallelen Speicherliste 43
2.9 Schnittstelle des Taskpool-Backends 44
2.10 Schnittstelle des Taskpool-Frontends 46
2.11 Vereinfachtes Beispiel einer taskbasierten Applikation 48

3.1 Pseudo-Code eines Tasks der synthetischen Applikation 50
3.2 Beispiel einer Taskerzeugung der synthetischen Applikation 50
3.3 Idealer Speedup des Quicksorts 58

4.1 Pseudo-Code der atomaren *Fetch & Op*-Operation 62
4.2 Pseudo-Code der atomaren *Compare & Swap*-Operation 62
4.3 Pseudo-Code der *Load & Reserve*- und *Store Conditional*-Operationen . . . 63
4.4 Implementierung des einfachen Locks für den SPARC-Prozessor 66
4.5 Implementierung des einfachen Locks für den Power4+-Prozessor 67
4.6 Implementierung des Ticket-Locks für den SPARC-Prozessor 68
4.7 Implementierung des Ticket-Locks für den Power4+-Prozessor 68
4.8 Implementierung des sperrfreien Listenzugriffs für den SPARC-Prozessor . . 70
4.9 Implementierung des sperrfreien Listenzugriffs für den Power4+-Prozessor . 71
4.10 Sequenzielle Ausführungszeit der synthetischen Applikation mit zentralen Taskpools 73
4.11 Parallele Ausführungszeit der synthetischen Applikation mit zentralen Taskpools 73
4.12 Sequenzielle Ausführungszeit der synthetischen Applikation mit verteilten Taskpools 79
4.13 Parallele Ausführungszeit der synthetischen Applikation mit verteilten Taskpools 80
4.14 Sequenzielle Ausführungszeit der synthetischen Applikation mit blockverteilten Taskpools 82
4.15 Parallele Ausführungszeit der synthetischen Applikation mit blockverteilten Taskpools 83
4.16 Speedups der Radiosity-Applikation mit zentralen Taskpools 84

4.17 Speedups der Radiosity-Applikation mit verteilten Taskpools 85
4.18 Speedups der Radiosity-Applikation mit blockverteilten Taskpools 85
4.19 Zusammenfassung der Speedups der Radiosity-Applikation 86
4.20 Speedup der parallelen Quicksort-Implementierung 87
4.21 Speedup der Volume-Rendering-Applikation 88
4.22 Speedup der Ray-Tracing-Applikation 90

5.1 Datenstruktur zur Speicherung von Speicherelementen eines Threads 97
5.2 Realisierung der Pthreads-Lock-basierten Lock-Komponente 99
5.3 Realisierung der Pthreads-Spinlock-basierten Lock-Komponente 100
5.4 Verwendete Datenstruktur für den Hardware-Lock der IA-64-Architektur . 101
5.5 Realisierung des Hardware-Locks auf der IA-64-Architektur 102
5.6 Alternative Realisierung des Hardware-Locks auf der IA-64-Architektur . . 102
5.7 Verwendete Datenstruktur für den Hardware-Lock der x86-Architektur . . . 103
5.8 Realisierung des Hardware-Locks auf der x86-Architektur 104
5.9 Verwendete Datenstruktur für den Hardware-Ticket-Lock 104
5.10 Realisierung des Hardware-Ticket-Locks auf der IA-64-Architektur (GCC) . 105
5.11 Realisierung des Hardware-Ticket-Locks auf der IA-64-Architektur (ICC) . 105
5.12 Realisierung des Hardware-Ticket-Locks auf der x86-Architektur 106
5.13 Verwendete Datenstruktur für den Duolock 107
5.14 Realisierung des Duolocks aus Sicht des Lock-Eigentümers 107
5.15 Realisierung des Duolocks aus Sicht der externen Threads 108
5.16 Interface der Hardware-Barrier-Unterstützung 109
5.17 Implementierung der Code-Barrier-Unterstützung 110
5.18 Implementierung der Mem-Barrier-Unterstützung 111
5.19 Datenstruktur der Speicherliste „memlist“ 112
5.20 Zugriff auf das Zeitregister in der x86-Architektur (64-Bit-Variante). 113
5.21 Zugriff auf das Zeitregister in der Power-Architektur (64-Bit-Variante). . . . 113
5.22 Zugriff auf das Zeitregister in der IA-64-Architektur. 114
5.23 Auszug eines Kalibrierungslaufes auf einer Itanium2-Architektur 119
5.24 Pseudo-Code-Darstellung der Taskabarbeitungsschleife 124

6.1 Baumdatenstruktur (schematisch) 128
6.2 Beispiel eines teilgefüllten Waldvektors 128
6.3 Einfügeoperation eines neuen Tasks in den Waldvektor 129
6.4 Beispiel für das Einfügen eines neuen Tasks T 130
6.5 Entnahmeoperation eines Tasks aus einem Waldvektor 130
6.6 Beispiel für die Entnahme eines Tasks 131
6.7 Task-Stealing-Operation eines Baums von Thread p_1 zu p_2 132
6.8 Beispiel für die Stehleoperation von Thread 2 133
6.9 Beispiel einer Unterteilung des Waldvektors in einen privaten und öffentlichen Bereich 136
6.10 Speedups der synthetischen Applikation für leere Tasks ($f = 0$) mit adaptiven Taskpools 138

6.11 Speedups der synthetischen Applikation für kleine Tasks ($f = 10$) mit adaptiven Taskpools . . . 139
6.12 Speedups der Quicksort-Applikation mit adaptiven Taskpools . . . 140
6.13 Speedups der Ray-Tracing-Applikation mit adaptiven Taskpools . . . 141
6.14 Speedups der hierarchischen Radiosity-Applikation mit adaptiven Taskpools 141

7.1 Taskabarbeitungsschleife im Taskpool-Frontend mit Zeitmessung . . . 147
7.2 Taskhistogramm für den Tasktyp „Visibility“ der Radiosity-Applikation . . 151
7.3 Taskhistogramm für den Tasktyp „Average“ der Radiosity-Applikation . . . 154
7.4 Taskhistogramm für den Tasktyp „Ray“ der Radiosity-Applikation . . . 155
7.5 Taskhistogramm für den Tasktyp „Refinement“ der Radiosity-Applikation . 156
7.6 Taskhistogramm der Ray-Tracing-Applikation . . . 157
7.7 Taskhistogramm der Volume-Rendering-Applikation . . . 158
7.8 Taskhistogramm der Quicksort-Applikation . . . 159
7.9 Beispielextrapolation für gegebene Messwerte . . . 162
7.10 Exemplarische Extrapolation für $pp = 16$ . . . 166
7.11 Auszug einer False-Sharing-Analyse der Radiosity-Applikation . . . 173
7.12 Speedups der verbesserten Radiosity-Applikation . . . 174
7.13 Speedups der verbesserten Radiosity-Applikation (zusätzlich balanciert) . . 175

Tabellenverzeichnis

3.1 Anzahl insgesamt erzeugter Tasks der synthetischen Applikation für bestimmte Taskargumente 51

4.1 Unterstützte Synchronisationsoperationen der betrachteten Systeme 64
4.2 Speedup und Effizienz der zentralen Taskpools für $f = 50$ 74
4.3 Speedup und Effizienz der verteilten Taskpools für $f = 50$ 81
4.4 Speedup und Effizienz der blockverteilten Taskpools für $f = 50$ 84
4.5 Speedups der nicht adaptiven Volume-Rendering-Applikation 89
4.6 Speedups und Taskgrößen der betrachteten Applikationen 91

7.1 Beispiele der Extrapolationsgewichte für ausgesuchte Messpunkte 165
7.2 Vorhergesagte und tatsächliche Ausführungszeit und Wartezeit für die Radiosity-Applikation 168

A.1 Technische Parameter des IBM p690 Systems (erste Ausbaustufe) 181
A.2 Technische Parameter des IBM p690 Systems (zweite Ausbaustufe) 181
A.3 Technische Parameter des SGI Altix 4700 Systems 182
A.4 Technische Parameter des Sun Fire 6800 Systems 182

Kapitel 1

Einleitung

Parallelrechner zeichnen sich dadurch aus, dass sie eine große Anzahl von Berechnungseinheiten besitzen, mit denen sie mehrere Größenordnungen schneller rechnen können als konventionelle Computer mit nur einem Prozessor. Solche Supercomputer setzen sich meist aus vielen einzelnen Prozessoren zusammen, die teilweise auf gemeinsamen Motherboards eingesetzt werden und mit einem gemeinsamen Bus entsprechend schnell miteinander kommunizieren können. Größere Systeme werden durch die Verbindung vieler Boards mit entsprechend schnellen Verbindungsnetzwerken gebaut. Der Hauptspeicher ist auf solchen Systemen auf vielen Motherboards verteilt und wird entweder implizit durch entsprechende Hardware allen anderen Prozessoren durch komplexe Speicherkohärenzprotokolle verfügbar gemacht oder muss explizit durch Nachrichtenaustausch zugegriffen werden. Seit einigen Jahren integrieren auch die bedeutendsten Chiphersteller wie Intel oder AMD in den Großteil ihrer Prozessorserien eine steigende Zahl von Prozessorkernen in einen einzelnen Prozessorchip, dem sogenannten Multi-Core-Prozessor. Um die mögliche Rechenkapazität dieser eng gekoppelten Prozessorkerne optimal auszunutzen, müssen neue Wege der Programmierung gefunden werden.

Auf Systemen mit verteilten Speichern müssen Zugriffe auf Daten anderer Prozessoren mit expliziten Nachrichten durchgeführt werden, was die Programmierung erschwert. Auf diesen Systemen wird häufig „MPI" (Message Passing Interface [93, 94]) eingesetzt. Die zur Kommunikation verschickten Nachrichten verzögern den Zugriff, aber auch auf SMP-Systemen mit gemeinsamem Speicher sind Speicherzugriffe auf gemeinsame Daten oft mit hohen Wartezeiten verbunden. Häufig teilen sich moderne Multi-Core-Prozessoren Caches und sind sehr schnell an den Hauptspeicher angebunden, sodass der Datenaustausch mit wesentlich weniger Latenzzeit verbunden ist.

Für die parallele Programmierung der eng gekoppelten Prozessorkerne können konventionelle Programmiertechniken angewendet werden. Um die maximale Performance zu erreichen, sollten jedoch die speziellen Eigenschaften dieser Prozessoren gezielt berücksichtigt werden. Insbesondere kann mittlerweile nicht mehr von einer gleichmäßig verteilten Rechenleistung der einzelnen Prozessorkerne ausgegangen werden. Durch Techniken wie *simultanes Multi-Threading* (SMT) [52, 128] oder dynamische Taktfrequenzanpassung kann die Abarbeitungsgeschwindigkeit zwischen baugleichen Prozessorkernen unterschiedlich sein, aber auch die Geschwindigkeit eines Kerns kann sich in relativ kurzen zeitlichen Abständen erheblich ändern.

Da die zugeordnete Rechenleistung eines Prozessors nicht mehr genau vorhersagbar ist, bietet ein feingranularer Berechnungsansatz mit dynamischer Lastbalancierung Potenzial zur Ausnutzung der Rechenkapazität.

Diese Arbeit konzentriert sich auf die Ausführung irregulärer Applikationen, deren Berechnungsstruktur und -aufwand nicht statisch feststeht, sondern erst zur Laufzeit festgelegt wird. Einerseits können bei irregulären Applikationen zwar die Art der Berechnungen, nicht aber deren Umfang bekannt sein. Andererseits können dynamische Datenstrukturen eine Loslösung von einer festen Berechnungsstruktur erfordern, sodass die Verteilung der parallelen Berechnungen entsprechend erschwert wird. Ein Beispiel einer solchen irregulären Applikation ist die Simulation von räumlich begrenzten Wechselwirkungen verschiedener Teilchen. In Gebieten hoher Teilchendichte ist eine höhere Anzahl von Wechselwirkungen zu berechnen als in Gebieten mit einer geringeren Dichte. Erst mit den Eingabedaten ist jedoch die Teilchendichte bekannt, die sich zudem im Verlauf der Simulation ändern kann. Aber selbst für reguläre Applikationen kann der feingranulare Abarbeitungsansatz Vorteile bieten, wenn z. B. die verwendete Rechnerarchitektur keine homogenen Berechnungseinheiten besitzt.

1.1 Ziele und Gliederung

Der Ansatz der *taskbasierten Programmierung* basiert auf einer Zerlegung der Berechnungen einer Applikation in einzelne Arbeitseinheiten, die sogenannten *Tasks*, die parallel zueinander abgearbeitet werden können. Die ausführungsbereiten Tasks werden in einer geeigneten Datenstruktur gespeichert, der sogenannte *Taskpool*, aus dem Prozessoren nach Bedarf Tasks zur Ausführung entnehmen können. Sind zu jedem Zeitpunkt ausreichend viele Tasks verfügbar, können alle Prozessoren beschäftigt werden, und die Applikation wird entsprechend schnell ausgeführt. Die Parallelisierung erfolgt somit nicht explizit für eine gegebene Anzahl von Prozessoren, sondern vielmehr unabhängig von dieser durch eine Zerlegung in möglichst viele parallel ausführbare Tasks. Bei modernen Multi-Core-Prozessoren sinkt die Zugriffs- und Synchronisationszeit zwischen den Kernen eines Prozessors gegenüber konventionellen Multi-Prozessor-Systemen, sodass sich ein taskbasierter Ansatz auch für Tasks eignet, die eine relativ kurze Ausführungszeit haben.

Die vorgelegte Arbeit untersucht, wie solche feingranularen Tasks effizient abgearbeitet werden können. Bei der Verwendung feingranularer, also kurz laufender Tasks nimmt der Overhead der Taskverwaltung einen besonderen Stellenwert ein. Eine Laufzeitumgebung zur taskbasierten Abarbeitung muss die Tasks sehr schnell den Prozessoren zur Ausführung zuweisen, damit die parallele Ausführung gute Speedups erreicht. Allerdings reicht es nicht, nur den Verwaltungsoverhead zu reduzieren. Je mehr Prozessoren an der Berechnung teilnehmen, desto schwieriger wird es, alle Prozessoren mit Tasks zu versorgen. Dynamische Lastbalancierungsverfahren verteilen die Last in Form der ausführungsbereiten Tasks möglichst gleichmäßig auf die Prozessoren. Da der mit einem Task verbundene Rechenaufwand bei irregulären Applikationen nicht im Voraus bekannt ist, muss die Verteilung während der Laufzeit erfolgen. Die Lastverteilung erzeugt jedoch zusätzlichen Overhead, sodass einerseits schnelle Verfahren gesucht sind, aber andererseits möglichst wenig in die Taskabarbeitung eingegriffen werden sollte.

Gerade bei einer großen Anzahl von Tasks kann es während der Abarbeitung häufig notwendig sein, sehr viele Tasks zwischen Prozessoren auszutauschen. Eine effiziente Lastbalancierung sollte diese Situationen mit wenig Overhead handhaben können. Insbesondere

bei Applikationen mit einer sehr ungleichen Lastverteilung und einer großen Anzahl von dynamisch erzeugten Tasks sollte das Balancierungsverfahren nicht ständig mit einer Umverteilung beschäftigt sein. Die eventuell nötigen Synchronisationsoperationen bei der Verteilung der Tasks zwischen Prozessoren können den Aufwand der Lastbalancierung erheblich erhöhen.

In dieser Arbeit soll deshalb eine adaptive Lastbalancierung entwickelt werden, um eine große Menge ausführungsbereiter Tasks effizient ausführen zu können. Die Implementierung sollte auch in solchen Situationen gute Ergebnisse erzielen, für die konventionelle Verfahren weniger geeignet sind.

Weiterhin soll eine Laufzeitumgebung entwickelt werden, um damit beliebige taskbasierte Applikationen implementieren zu können. Die Umgebung soll die Möglichkeit bieten, verschiedene Lastbalancierungsverfahren nutzen und vergleichen zu können.

Erzielt eine taskbasierte Applikation bei der parallelen Ausführung nicht die erwarteten Laufzeiten, müssen oft Hilfsmittel zur Problemanalyse verwendet werden. Mit Profiling-Tools können problematische Programmstellen innerhalb der Applikation entdeckt werden. Bei taskbasierten Programmen können sich Tasks derart gegenseitig behindern, dass eine parallele Abarbeitung kaum effizient möglich ist. Zudem können nicht nur Probleme innerhalb der Applikation existieren, sondern auch innerhalb des Taskpools. Um die Analyse einer taskbasierten Applikation zu erleichtern, soll ein Profiling-Mechanismus entworfen werden, mit dem die Taskstruktur einer Applikation analysiert und Skalierbarkeitsprobleme identifiziert werden können.

Im folgenden Kapitel 2 dieser Arbeit wird eine Einführung in die taskbasierte Programmierung gegeben und das komponentenbasierte KOALA-Framework beschrieben. Für Laufzeitexperimente werden Applikationen mit unterschiedlichen Charakteristika bei der parallelen Abarbeitung betrachtet, die in Kapitel 3 beschrieben werden. In Kapitel 4 wird untersucht, wie sich der Aufwand für die Verwaltung feingranularer ausführungsbereiter Tasks durch Nutzung von Hardware-Operationen verschiedener Plattformen verringern lässt. Das Kapitel 5 beschreibt die verfügbaren Implementierungen der Framework-Komponenten. In Kapitel 6 wird eine adaptive Datenstruktur und die zugehörige Implementierung eines adaptiven Taskpools präsentiert. Da oft eine betrachtete Applikation im praktischen Einsatz nicht das erwünschte Skalierungsverhalten zeigt, wird in Kapitel 7 eine Framework-Komponente vorgestellt, mit der eine genaue Analyse der taskbasierten Abarbeitung möglich ist. Die Ergebnisse der Arbeit werden in Kapitel 8 zusammengefasst.

1.2 Begriffsklärung

1.2.1 Zielarchitekturen

Supercomputer setzen sich aus einer Vielzahl von Prozessoren zusammen, die die verfügbare Rechenleistung vervielfachen sollen. Ein wesentliches Unterscheidungsmerkmal ist das eingesetzte Speichersystem. Auf Systemen mit *verteiltem Speicher* muss der Programmierer mit Hilfe expliziter Kommunikationsoperationen auf Daten im Speicher anderer Prozessoren zugreifen, z. B. mit Hilfe der MPI-Bibliothek [93, 94]. Dies erschwert die Programmierung, da bei jeder Operation Quelle und Ziel der Kommunikation bekannt sein und die Daten explizit bereitgestellt werden müssen. Ein Vorteil ist jedoch, dass entsprechende Operationen

gezielt geplant und teilweise überlappend mit Berechnungen ausgeführt werden können.

Systeme mit *gemeinsamem Speicher* (Shared-Memory-Systeme) benutzen dagegen einen globalen Adressraum, in dem jedes Speicherelement der verschiedenen Prozessoren abgebildet ist. Somit ist aus Sicht des Prozessors zwischen eigenem, lokal verfügbaren Speicher und fremdem Speicher anderer Prozessoren kein Unterschied sichtbar. Entsprechend können Datenblöcke verwendet werden, die wesentlich größer sind als im Speicher eines einzelnen Prozessors abgespeichert werden können. Jede beliebige Adresse kann von einem Prozessor direkt geladen werden.

Die einzelnen Prozessoren und deren Speicherelemente sind je nach Architektur durch Bussysteme oder Verbindungsnetzwerke miteinander verbunden. Die Hardware des Speichersystems stellt eine konsistente Sicht der globalen Daten sicher. Ein Merkmal einer solchen Speicherimplementierung ist jedoch, dass Zugriffe auf entfernten Speicher mehr Zeit benötigen als Zugriffe auf den eigenen Speicher. Der Speicher eines Prozessors kann z. B. über ein schnelles Bussystem angeschlossen sein, während andere Prozessoren über ein Netzwerk zugegriffen werden müssen.

Ist der Unterschied der Zugriffszeiten nicht existent oder nur marginal, spricht man von UMA-Systemen (Uniform Memory Access). Bei NUMA-Systemen (Non-Uniform Memory Access) sind dagegen die Zugriffszeiten unterschiedlich groß.

In dieser Arbeit werden Systeme mit gemeinsamem Speicher betrachtet, die allgemein als SMP-Systeme bezeichnet werden (Symmetric Multiprocessing). Solche Systeme sind in einer Vielzahl unterschiedlicher Konfigurationen verfügbar, wobei die aktuelle Entwicklung eine stark ansteigende Anzahl der verfügbaren Recheneinheiten prognostiziert. Laut einer Studie von Intel [16] könnte im Jahre 2015 ein einzelner Prozessorchip aus Hunderten von Prozessorkernen bestehen, die dabei nicht unbedingt mehr einen homogenen Aufbau haben müssen. Da die Taktraten schon aus thermischen Gründen mittlerweile weniger stark ansteigen [23, 101], werden eng gekoppelte Prozessorkerne sehr häufig anzutreffen sein. Die Ausnutzung der auf diese Art gebotenen Rechenkapazitäten ist eine große Herausforderung für zukünftige Parallelisierungsmechanismen.

Mittlerweile sind selbst im Endanwenderbereich Prozessoren mit zwei, vier oder sogar acht Kernen die Regel. Entsprechend existieren Parallelrechner, die eine große Anzahl solcher Prozessoren in einem System verbinden. Zusätzlich erlauben Technologien wie simultanes Multi-Threading (SMT) [52, 128] die bessere Ausnutzung der verfügbaren Rechenkapazitäten eines einzelnen Prozessorkerns. Dabei wird ein Prozessorkern durch virtuelle Prozessoren erweitert. Jeder virtuelle Prozessor besitzt einen eigenen Registersatz mit Programmzähler, Statusregistern und Datenregistern, die jeweils den Zustand eines Prozessors widerspiegeln. Alle virtuellen Prozessoren teilen sich aber die Funktionseinheiten und Caches des Prozessorkerns. Eine entsprechende Logik innerhalb des Prozessors weist den virtuellen Prozessoren die Funktionseinheiten entsprechend ihres jeweils ausgeführten Programmcodes zu. Eine Einheit kann nur von einem virtuellen Prozessor benutzt werden, es können aber gleichzeitig mehrere virtuelle Prozessoren unterschiedliche Funktionseinheiten belegen. Dadurch können die Einheiten besser ausgenutzt werden, insbesondere dann, wenn unabhängige Threads abgearbeitet werden oder der Programmcode nicht optimal für die Zielarchitektur übersetzt wurde. Da die Registersätze im Vergleich zu vollwertigen Kernen relativ wenig Chipfläche benötigen, kann ohne große Kosten die Ausnutzung der Funktionseinheiten verbessert werden. Allerdings kann durch gegenseitige Behinderung der virtuellen

Prozessoren auch eine Verlangsamung eintreten.

Die u. a. für diese Arbeit betrachtete IBM-Maschine „JUMP“ des Forschungszentrums Jülich setzte ursprünglich Power4+-Prozessoren ein, die jeweils zwei Kerne besaßen. Eine spätere Ausbaustufe setzt die aktuellen Power6-Prozessoren ein, die ebenfalls zwei Kerne besitzen, allerdings zusätzlich SMT mit zwei Threads pro Kern unterstützen.

Der Ansatz der VLIW-Architektur [52] (Very Long Instruction Word) verfolgt einen anderen Weg der Ausnutzung der Funktionseinheiten eines Prozessors. Bei diesem Ansatz kann der Compiler direkt die einzelnen Funktionseinheiten durch entsprechende Codeerzeugung ansprechen, sodass Kontrolllogik auf Seiten des Prozessors eingespart werden kann und höhere Geschwindigkeiten erreicht werden können. Da der Compiler einerseits im Vergleich zum Prozessor fast unbegrenzt Zeit verwenden kann, um eine optimale Ablauffolge zu bestimmen, und andererseits auch wesentlich mehr Informationen zur Compile-Zeit zur Verfügung steht, kann der erzeugte Code effizienter abgearbeitet werden. Intel hat diesen Ansatz unter dem Namen EPIC-Design (Explicitly Parallel Instruction Computing [112]) in den Itanium-Prozessor implementiert. Für diese Arbeit wird auch ein Parallelrechner mit solchen Prozessoren betrachtet. Die SGI-Altix-Maschine [34] des LRZ München besitzt insgesamt 9728 Itanium2-Prozessorkerne, die über eine NUMAflex-Architektur verbunden sind und einen gemeinsamen Adressraum bieten. Die Zugriffszeiten sind jedoch sehr stark abhängig vom Abspeicherungsort der zugegriffenen Daten. Diese NUMA-Architektur stellt zusätzliche Herausforderungen an eine effiziente Programmierung.

Als Forschungsstudie hat Intel mit dem Terascale-Prozessor [50] einen Entwicklungschip mit 80 Kernen präsentiert, dessen Kerne einfache, aber vollständige Funktionseinheiten beinhalten. Für diesen „Many-Core“-Prozessor ist eine komplexe Cache-Hierarchie vorgesehen mit eventuell zusätzlichen kleinen lokalen Speichern. Genaue Implementierungsdetails sind jedoch nicht öffentlich verfügbar.

Einen komplett anderen Ansatz verfolgt dagegen die Cell-Architektur [59], die neben einem herkömmlichen Prozessorkern basierend auf der Power-Architektur auch (derzeit) acht Kerne als SIMD-Prozessoren besitzt (sogenannte SPUs, vgl. [109]). Diese heterogene Architektur bietet eine hohe Performance, erfordert aber eine geeignete Programmierung. Die Performance hängt dabei wesentlich von einer effektiven Nutzung der lokalen Speicher [35] der SPUs ab. Diese Speicher sind ähnlich schnell wie Caches, sind allerdings per Software zu steuern und erlauben einen flexibleren Einsatz dieses Speichers.

Der UltraSPARC-T2-Prozessor [117] von SUN ist ebenfalls schon seit einiger Zeit verfügbar und besitzt 8 Prozessorkerne, die jeweils 8 Threads auf Basis von SMT ausführen können. Für solche Systeme mit 64 virtuellen Kernen kann eine dynamische Lastbalancierung auf Basis taskbasierter Abarbeitung gute Resultate liefern, da auf die unterschiedliche Abarbeitungsgeschwindigkeit der einzelnen Threads reagiert werden kann.

Auch für Systeme mit verteiltem Speicher kann ein gemeinsamer Adressraum simuliert werden, um die Vorteile der einfacheren Programmierung ausnutzen zu können. Dazu werden Softwarelösungen verwendet, die als *Distributed Shared Memory* (DSM) bezeichnet werden (siehe [41, 96, 114] sowie [106] für einen einführenden Überblick). Der Aufwand für die softwareseitige Verwaltung eines gemeinsamen Adressraums ist jedoch relativ hoch, da Zugriffe auf Daten anderer Prozessoren gegebenenfalls abgefangen und entsprechende Kommunikationsoperationen ausgeführt werden müssen.

In dieser Arbeit werden jedoch ausschließlich Systeme mit gemeinsamem Speicher be-

trachtet. Details zu den verwendeten Rechnersystemen auf Basis von Itanium2-Prozessoren, Power-Prozessoren und SPARC-Prozessoren sind in Anhang A aufgelistet.

1.2.2 Parallele Ausführungsumgebung

Die parallele Ausführung einer Applikation auf Systemen mit gemeinsamem Speicher kann auf verschiedene Arten durchgeführt werden. Programme werden aus Sicht des Betriebssystems als Prozesse verwaltet, die einen eigenen virtuellen Adressraum besitzen und so klar von anderen Prozessen getrennt sind. Das Betriebssystem sorgt durch einen Scheduler dafür, dass die ausführungsbereiten Prozesse an die freien Prozessoren zugewiesen werden. Entsprechend kann eine Applikation mehrere Prozesse erzeugen, um auf verschiedenen Prozessoren parallele Berechnungen durchführen zu können. Aufgrund des getrennten Adressraums der Prozesse muss auf Systemen mit gemeinsamem Speicher ein expliziter Nachrichtenaustausch durchgeführt werden (Interprozesskommunikation).

Daher wird auf solchen Systemen häufiger auf Threads zurückgegriffen, welche aus Sicht des Betriebssystems leichtgewichtige Prozesse darstellen und ähnlich wie normale Prozesse an die freien Prozessoren zugewiesen werden. Ein Prozess kann mehrere Threads erzeugen, die alle den gleichen virtuellen Adressraum besitzen, sodass alle parallel laufenden Threads auf die gemeinsamen Daten direkt zugreifen können. Jeder Thread besitzt einen eigenen Kontrollfluss und kann somit jeweils ein unterschiedliches Teilprogramm ausführen. Für die Programmierung solcher Programme werden in der Praxis verschiedene Thread-APIs (*Application Programming Interface*) wie POSIX-Threads (Pthreads) [22] oder Java-Threads [99] eingesetzt.

In dieser Arbeit wird die feingranulare Abarbeitung paralleler Programme betrachtet, d. h. das Programm wird in viele kleine, parallel ausführbare Teile zerlegt, die als *Tasks* ausgeführt werden. Prinzipiell kann bei einer solchen taskbasierten Programmabarbeitung jeder Task einer Applikation durch einen separaten Thread ausgeführt werden [97]. Bei einer hohen Anzahl von Threads, wie sie für feingranulare Applikationen erforderlich ist, führt dabei die Abbildung auf im Zeitscheibenverfahren arbeitende Betriebssystemthreads allerdings zu einem hohen Overhead für Erzeugung und Verwaltung der Threads und Kontextwechsel der Betriebssystemthreads sowie zu einem hohen Ressourcenbedarf für die Speicherung von Registern und Stack-Inhalten. Existieren auf diese Art deutlich mehr Threads als Prozessoren, so wird vom Betriebssystem eine zyklische Zuteilung mehrerer Threads auf jeweils einen Prozessor durchgeführt. Dadurch kann es zu einer schlechteren Ausnutzung der Lokalität kommen, da sich die Threads entsprechend auch die Caches teilen müssen. Nicht unterbrechbare Threads (z. B. Nano-Threads [103]) eignen sich für eine Verringerung des Ressourcenverbrauchs, da deren Ressourcen vom jeweils nachfolgend ausgeführten Thread direkt weiterverwendet werden können [131] und Kontextwechsel nur beim Blockieren oder bei Beendigung eines Threads erfolgen.

Viele Programmierumgebungen abstrahieren diese Thread-Schicht und stellen dem Programmierer Mittel bereit, die Parallelität innerhalb des Programms geeignet zu deklarieren, während die eigentliche Zuteilung an Betriebssystemthreads vom Programmierer weitgehend versteckt wird. Beispielsweise stellen Bibliotheken wie Intels „Threading Building Blocks“ (TBB) [111] oder die „Multi-Core Standard Template Library“ [121] Softwarebausteine sowohl für parallele Algorithmen (z. B. Suchen und Sortieren) als auch einfache

Konstrukte für die parallele Ausführung von Programmteilen nach einem vorgegebenen Muster (z. B. parallele Schleifen, Reduktionsoperationen und Pipelines) bereit. TBB nutzt Hardware-Operationen für Synchronisationsmechanismen, ist allerdings derzeit nur auf einer kleinen Zahl von Architekturen verfügbar.

Die Spracherweiterung OpenMP [102] definiert eine Reihe von Quellcode-Annotationen für verschiedene sequenzielle Programmiersprachen, die sich hauptsächlich für die Parallelisierung von Schleifen eignen. In den Intel-OpenMP-Compiler wurde allerdings eine *taskq*-Anweisung integriert, die eine taskbasierte Beschreibung des Anwendungsalgorithmus erlaubt, vgl. auch [125]. In den Standard OpenMP 3.0 wurde eine entsprechende Taskdirektive aufgenommen. Allerdings sind bei beiden Erweiterungen die Einflussmöglichkeiten des Programmierers auf die Art der Lastbalancierung sehr gering.

Andere Ansätze führen komplett neue Programmiersprachen ein oder erweitern vorhandene Sprachen durch aufwendigere Konstrukte zur Steuerung der parallelen Abarbeitung. Die Programmiersprache Cilk [12] und die Bibliothek DOTS [11] schlagen für die Beschreibung paralleler Programme ein *fork-join*-Modell für striktes Multi-Threading vor. Charm++ [65] ist eine auf C++ basierende parallele objektorientierte Sprache, die Lastbalancierung durch Migration von Objekten und *„Seed Load Balancing“* unterstützt. Andere Umgebungen wie RAPID [139] oder PYRROS [43] führen eine statische Ablaufplanung auf Basis gerichteter azyklischer Taskgraphen (DAGs) durch. VDS [31] unterstützt verschiedene Programmierparadigmen wie striktes Multi-Threading, unabhängige Tasks und statische DAGs. Allerdings erfordert das System einen hohen Programmieraufwand auf Seiten der Applikation.

Als Erweiterung der Programmiersprache C für große parallele Systeme wurde Unified Parallel C (UPC [25, 36]) vorgeschlagen. UPC ist auf dem Modell eines gemeinsamen, partitionierten Adressraums aufgebaut, auf den jeder Prozessor zugreifen kann. Jede Variable ist jedoch physikalisch einem einzelnen Prozessor zugeordnet [72]. Ein Task-Konzept wird jedoch nicht unterstützt. Die Programmiersprache Sequoia [39], die ebenfalls auf der Programmiersprache C basiert, wurde mit dem Ziel einer bestmöglichen Ausnutzung der Speicherhierarchie der jeweiligen Hardware konzipiert. Durch ein statisches und taskbasiertes Programmiermodell soll es dem Programmierer ermöglicht werden, die Platzierung und den Austausch von Daten innerhalb der Speicherhierarchie zu kontrollieren.

Die Sprache Chapel [29] erlaubt die Ausnutzung von Datenparallelität, z. B. in Form von *forall*-Schleifen. Außerdem stehen verschiedene Synchronisationsmechanismen wie z. B. *atomic*-Blöcke zur Verfügung. Eine explizite taskbasierte Programmbeschreibung ist allerdings in der aktuellen Sprachspezifikation nicht vorgesehen.

Sowohl Datenparallelität, z. B. auf Basis von parallelen *for*-Schleifen, als auch Taskparallelität werden in den Sprachen Fortress [4] und X10 [27] unterstützt. Beide bieten außerdem Synchronisationsmechanismen wie einen *atomic*-Block.

Innerhalb dieser Arbeit wird ein taskbasierter Ansatz verfolgt, bei dem mehrere Threads benutzt werden, um feingranulare Berechnungseinheiten auszuführen. Dabei werden einem Thread wiederholt Tasks zur Ausführung zugewiesen. Die Verwaltung der Threads und die Zuweisung an die Prozessoren wird also dem Betriebssystem überlassen, die optimale Ausnutzung der Threads wird allerdings von einer eigenen Laufzeitumgebung sichergestellt. Wenn nicht anders angegeben, wird daher auch jeweils ein Thread für einen zu benutzenden Prozessor verwendet, sodass diese 1:1-Zuordnung wenig Verwaltungsoverhead auf Seiten des Betriebssystems erfordert. Eine genauere Einführung wird in Kapitel 2 gegeben.

Die taskbasierte Implementierung paralleler Applikationen wurde ursprünglich für die effiziente Realisierung irregulärer Algorithmen auf (homogenen) Parallelrechnern mit gemeinsamem Adressraum vorgeschlagen (siehe z. B. [120]). Für neuere Hardware-Entwicklungen wie den Cell-Prozessor schlägt der Hersteller IBM u. a. eine taskbasierte Ausführung vor [105], sodass die Untersuchung sehr feingranularer Tasks weiter an Bedeutung gewinnt. In [9] wird eine taskbasierte Programmierumgebung CellSs für den Cell-Prozessor zur Ausnutzung von Funktionsparallelität vorgestellt. Auch Intel betrachtet die taskbasierte Programmierung als vielversprechenden Ansatz zur Ausnutzung des Potenzials von Multi-Core-Prozessoren [16] und erforscht deshalb Möglichkeiten der Hardware-seitigen Unterstützung für feingranulares Task-Scheduling (Carbon [74]). Ein taskbasiertes Programmiermodell ist auch Bestandteil von Intels Laufzeitbibliothek „Threading Building Blocks“ [111], die ein auf Task-Stealing beruhendes Lastbalancierungsverfahren verwendet.

Die Java-Plattform unterstützt ebenfalls ein einfaches taskbasiertes Programmiermodell für einen gemeinsamen Adressraum. Seit der Version 1.5 enthält sie das Paket *java.util.concurrent*, das mehrere Varianten sogenannter Thread-Pools implementiert [44]. Dieses Programmiermodell zielt jedoch eher auf Tasks größerer Granularität.

1.2.3 Lastbalancierung

Für eine effiziente parallele Programmabarbeitung müssen die Berechnungsteile des Programms geeignet auf die verfügbaren Berechnungseinheiten verteilt werden. Dies kann durch statisches Scheduling zur Übersetzungzeit des Programms erfolgen oder durch dynamisches Scheduling während der Abarbeitung. In dieser Arbeit wird die taskbasierte Programmabarbeitung betrachtet, bei der während der Laufzeit neue Tasks erzeugt werden und so eine dynamische Verteilung der Tasks notwendig ist. Dennoch soll hier ein kurzer Überblick über beide Ansätze folgen.

Statische Lastbalancierungsverfahren

Statisches Scheduling von Berechnungen wurde intensiv erforscht. Dabei wird angenommen, dass der komplette Taskgraph mit den Ausführungs- und Kommunikationskosten der einzelnen Tasks als DAG bekannt ist. Auf dieser Basis berechnen die statischen Verfahren einen Ablaufplan, mit dem später die Tasks effizient in der entsprechenden Reihenfolge abgearbeitet werden. Ein allgemeiner Überblick über statische Schedulingverfahren wird z. B. in [17] und [80] gegeben.

Grundlage für eine statische Verteilung von Berechnungen sind in der Regel Partitionierungstechniken für Graphen. Die Bestimmung einer optimalen Verteilung für mehr als zwei Prozessoren ist jedoch NP-hart [14, 15], sodass meist auf Heuristiken zurückgegriffen werden muss.

Statische Schedulingverfahren sind prinzipbedingt nicht direkt auf irreguläre und adaptive Berechnungen anwendbar. Für iterative adaptive Berechnungen können jedoch Repartionierungsverfahren eingesetzt werden [100, 141]. Mit PARTY [92], METIS und ParMETIS [66] sind entsprechende Bibliotheken verfügbar, die diese Techniken realisieren. In [90] werden graphbasierte Lastbalancierungen für FEM-Simulationen eingesetzt. In [71] wird ein auf Hypergraph-Partitionierungen basierendes Verfahren vorgestellt, das die aus-

zuführenden Tasks speziell aufgrund der Speicherhierarchie auswählt, um ausgehend von Informationen zu den Berechnungen eines Tasks und den benötigten Daten die nötigen Datentransferoperationen zu minimieren.

Sind die Ausführungszeiten der einzelnen Tasks nicht bekannt, kann nicht mehr für einen beliebigen Zeitpunkt die Menge der ausführungsbereiten Tasks angegeben werden. Dennoch können dafür teilweise statische Verfahren eingesetzt werden. In [126] wird die Ausführungszeit anhand von wahrscheinlichen Verteilungen modelliert, in [26] werden mit Hilfe von „Fuzzy Tasks" Optimierungen des Schedulings vorgeschlagen. In [42] wurde das statische Scheduling-System PYRROS [140] auf die „Fast Multipole"-Methode (FMM) angewendet.

Dynamische Lastbalancierungsverfahren

Statische Lastbalancierungsverfahren setzen die Kenntnis des Taskgraphen und oft eine fest definierte Ausführungsumgebung voraus. Für heterogene Plattformen und irreguläre Applikationen werden daher oft dynamische Schedulingverfahren für Einzelprozessortasks eingesetzt [138]. Der Einsatz von Diffusionsmethoden wurde z. B. für Branch-and-Bound-Verfahren untersucht [84, 127]. Eingesetzt werden auch randomisierte Varianten dieser Verfahren, die z. B. eine zufällige Auswahl der Prozessoren treffen, die an der Lastverteilung beteiligt werden, vgl. auch [37, 85].

Die für die Realisierung eines Lastbalancierungsverfahrens verwendeten Datenstrukturen und Zugriffsverfahren haben einen großen Einfluss auf die resultierende parallele Ausführungszeit der jeweils betrachteten Applikation [69]. Die verfügbaren Tasks können in zentralen oder verteilten Datenstrukturen gespeichert werden. Hierarchische Datenstrukturen [30] können den Aufwand bei der Suche nach verfügbaren Tasks reduzieren und auch bei einer hohen Anzahl von Prozessoren eine gute Skalierbarkeit bieten. Der Zugriff auf die Datenstrukturen und die Lastbalancierung kann auf verschiedene Arten realisiert werden [76]. Empfängerbasierte Verfahren führen die Balancierung dann aus, wenn ein Prozessor nicht ausgelastet ist. Dieser sucht dann, z. B. bei anderen Prozessoren, nach verfügbaren Tasks und führt sie aus (*task stealing* [20, 47]). Bei senderbasierten Lastbalancierungsverfahren versucht dagegen ein überlasteter Prozessor, Arbeit an andere Prozessoren abzugeben. Beispielsweise wird für die Realisierung paralleler Garbage-Collections eine Lastbalancierungsstrategie namens *task pushing* vorgeschlagen, die ohne Synchronisationsoperationen realisiert werden kann [137]. Eine weitere Möglichkeit besteht darin, die Programmausführung in periodischen Abständen zur Durchführung der Lastbalancierung zu unterbrechen.

Das Taskmodell kann auch durch Multiprozessortasks (*M-Tasks*) erweitert werden, mit denen mehrere Prozessoren zur Ausführung eines einzelnen M-Tasks benutzt werden können. Ansätze für das Scheduling solcher M-Tasks werden z. B. in [63, 79, 129] betrachtet. Aufgrund der Kosten für M-Task-interne Kommunikation und Verwaltung sind sie für feingranulare Berechnungen jedoch weniger geeignet.

Können Berechnungen als parallele Schleifen mit unabhängigen Iterationen formuliert werden, ist eine Verteilung der Schleifeniterationen auf die vorhandenen Prozessoren mit Hilfe von Loop-Schedulingverfahren [40] möglich. Die Schleifeniterationen werden dazu einzeln oder in Gruppen unter Wahrung der Schleifenabhängigkeiten auf die vorhandenen Prozessoren verteilt. Bei festen Schleifengrenzen und bekanntem Berechnungsaufwand der einzelnen Iterationen kann eine Verteilung statisch bestimmt werden. Andernfalls können

dynamische Verfahren wie Guided-Self-Scheduling [104] oder adaptive Faktorisierung [8] verwendet werden, um zur Laufzeit eine Lastbalancierung zu realisieren. Andere Ansätze verfolgen ein automatisches Loop-Scheduling ([24, 28, 110]). Dynamische Lastbalancierungsverfahren für parallele Schleifen sind zum Beispiel in OpenMP [102] oder High Performance Fortran (HPF) [55] implementiert. Ein System zur Durchführung von Schleifenparallelisierungen ist in [46] beschrieben.

In dieser Arbeit wird eine dynamische Lastbalancierung einzelner Tasks betrachtet, bei der ausführungsbereite Tasks möglichst effizient an die freien Prozessoren zugeordnet werden sollen. Die Menge der Tasks soll derart verwaltet werden, dass der Overhead für die Suche eines Tasks möglichst gering ist.

1.2.4 Leistungsbewertung

Für die Bewertung der Performance einer parallelen Applikation werden in dieser Arbeit im Wesentlichen die entsprechenden Laufzeiten ausgewertet, wobei drei Messwerte betrachtet werden. Die *Real-Zeit* stellt die tatsächlich vergangene Zeit dar. Da das primäre Optimierungsziel die schnellstmögliche Ausführung ist, wird diese Messgröße für Vergleichszwecke verwendet. Darüber hinaus gibt die *User-Zeit* die für Berechnungen im Programm verwendete Zeit an. Ausführungsunterbrechungen, wie z. B. durch I/O-Operationen oder Wartezeiten, sind in dieser Messgröße nicht erfasst. Für jeden Thread einzeln speichert das Betriebssystem die Ausführungszeiten, sodass die User-Zeit eines Prozesses die Summe der Ausführungszeiten aller Threads ist. Als dritte Messgröße gibt die *System-Zeit* an, wie viel Zeit in Betriebssystemsfunktionen vergangen ist.

Für die Bewertung einer parallelen Ausführung wird häufig der *Speedup* verwendet, der als Faktor die parallele Ausführungszeit t_{par} der sequenziellen Ausführungszeit t_{seq} gegenüberstellt:

$$\text{Speedup} = \frac{t_{seq}}{t_{par}},$$

wobei die beiden Messzeiten t_{seq} und t_{par} jeweils die Real-Zeiten darstellen. Die User-Zeit ist im Idealfall auch bei der parallelen Ausführung identisch zur sequenziellen Ausführung, da die Berechnung auf die verfügbaren Prozessoren verteilt wird und somit in der Summe wieder den Wert der sequenziellen Ausführung erreicht.

Entsprechend lässt sich auch die parallele Effizienz definieren, die die Güte der parallelen Ausführung angibt:

$$\text{Effizienz} = \frac{\text{Speedup}}{\#\text{Threads}},$$

wobei ein Wert von Eins die bestmögliche Effizienz darstellt.

Über diese allgemeinen Zeitmesswerte hinaus gibt es weitere Möglichkeiten, Performance-Kenngrößen zu ermitteln. In dieser Arbeit wird eine genaue und schnelle Zeitmessung auf Basis von Prozessorregistern betrachtet, die in Kapitel 5.3.3 genauer beschrieben wird.

1.2.5 Betrachtete Applikationsklasse

In dieser Arbeit werden im Speziellen irreguläre Applikationen und deren taskbasierte Abarbeitung betrachtet. Eine parallele Applikation kann dabei verschiedene Arten von Irregularität aufweisen. Irreguläre Kontrollstrukturen, wie z. B. Fallunterscheidungen, bedingte Verzweigungen oder variable Schleifengrenzen, können die Parallelisierung erschweren, da der Umfang der Berechnungen nicht mehr vollständig bekannt sein kann.

Die verwendeten Datenstrukturen in der Applikation können ebenfalls Irregularität aufweisen, z. B. durch nicht balancierte Bäume, sodass beispielsweise verschiedene Teilbäume unterschiedlich viele Knoten beinhalten und so eventuell unterschiedlich viele Operationen damit auszuführen sind.

Darüber hinaus kann Irregularität auftreten, wenn während der Synchronisation und Kommunikation mehrerer Threads unvorhersagbare Verzögerungen auftreten. Dies kann u. a. auch durch irreguläre Daten- und Kontrollstrukturen verstärkt werden, da z. B. der Zeitpunkt der Fertigstellung einer bestimmten Operation nicht mehr bekannt ist und somit unbestimmt lange auf dessen Ergebnis gewartet werden muss.

Irreguläres Verhalten kann also in verschiedenen Bereichen einer Applikation auftreten, die eine optimale Ausnutzung der parallelen Berechnungseinheiten entsprechend erschweren. Die Ausführungszeit einzelner Applikationstasks ist unter solchen Voraussetzungen schwer vorhersagbar. Mit einer dynamischer Lastbalancierung, wie sie in dieser Arbeit betrachtet wird, kann allerdings auf diese Irregularität reagiert werden, indem frei werdenden Prozessoren neue Aufgaben zugewiesen werden.

Solche irregulären Algorithmen sind in Applikationen aus verschiedenen Bereichen wissenschaftlicher Berechnungen zu finden. Zu nennen sind Applikationen aus der Computergrafik wie das hierarchische Radiosity-Verfahren [48, 119], Volume-Rendering [115] oder Ray-Tracing [133] sowie aus dem Bereich der N-Körper-Simulation wie die Barnes-Hut-Methode, die Fast-Multipole-Methode (FMM) [119] oder das Linked-Cell-Verfahren aus der Moleküldynamik [45]. Weitere Beispiele sind Berechnungen mit dünn besetzten Matrizen, Monte-Carlo-Simulationen, parallele Divide-&-Conquer-Algorithmen, Waveform-Relaxationsverfahren für gewöhnliche Differentialgleichungssysteme [19] sowie adaptive gitterbasierte Lösungsverfahren für partielle Differentialgleichungen [70].

Kapitel 2

Das Taskpool-Programmiermodell und das KOALA-Framework

Die entwickelten taskbasierten Lastbalancierungsalgorithmen werden in das komponentenbasierte Framework „KOALA“ integriert, dessen Bestandteile in diesem Kapitel erläutert werden. Grundlage der parallelen Programmabarbeitung stellt dabei eine taskbasierte Beschreibung der Applikation dar. Vor der Beschreibung des Frameworks und dessen Komponenten soll ein Überblick über dieses Programmiermodell gegeben werden.

2.1 Parallele Programmausführung mit Tasks

Die Beschreibung der parallelen Abarbeitung eines Programms kann durch verschiedene Mechanismen erfolgen. Im Wesentlichen sind dabei Datenparallelität, Taskparallelität und Threadparallelität zu nennen. Datenparallelität beschreibt eine SIMD-orientierte Bearbeitung der Eingabedaten (Single Instruction, Multiple Data). Verschiedene, je nach Applikation geeignet definierte Teilbereiche einer Datenstruktur werden vom gleichen Programmcode parallel berechnet. Beispielsweise können mehrere Prozessoren verschiedene Iterationen einer Schleife parallel ausführen (sofern keine Abhängigkeiten zwischen Iterationen bestehen), wobei jede Iteration einen anderen Datenbereich berechnet. Da nicht immer auf disjunkten Daten parallel gerechnet wird, erfolgt ein Datenaustausch durch entsprechende Methoden der Rechnerumgebung. Bei Systemen mit gemeinsamem Speicher (Shared-Memory-Systeme) kann ein Zugriff direkt erfolgen, eventuell ist ein wechselseitiger Ausschluss nötig. Auf Maschinen mit verteiltem Adressraum werden Nachrichten z. B. mit der MPI-Bibliothek [94] verschickt, um benötigte Daten zu erhalten oder zu verschicken.

Task- und Threadparallelität folgen dem MIMD-Berechnungsprinzip (Multiple Instructions, Multiple Data). Die verschiedenen Prozessoren bzw. Threads führen dabei unterschiedlichen Programmcode aus. Datenparallele Programme können auch taskparallel ausgeführt werden, indem für jeden Teilbereich einer Datenstruktur ein Task mit jeweils gleichem Programmcode definiert wird. Darüber hinaus erlaubt ein Taskansatz das Loslösen von einer vorgegebenen sequenziellen Programmstruktur, wodurch eventuell eine höhere Parallelisierbarkeit realisiert werden kann.

Diese Arbeit untersucht eine solche taskbasierte Programmausführung. Ein Task stellt dabei die kleinste Einheit im Sinne einer parallelen Abarbeitung dar. Allgemein definiert sich ein Task durch eine Folge von Instruktionen, der Menge von Eingabedaten und der Menge von Ausgabedaten, die während der Abarbeitung der Instruktionen des Tasks verarbeitet werden. Ein taskbasiertes Programm setzt sich entsprechend aus einer Menge von Tasks

zusammen, die eventuell in einer Abhängigkeitsrelation stehen.

Ein taskbasiertes Programm ist somit ein gerichteter azyklischer Graph (Directed Acyclic Graph, *DAG*). Ist dieser Graph vor der Ausführung bekannt, können statische Scheduling-algorithmen benutzt werden, um eine möglichst optimale Abarbeitungsreihenfolge für eine gegebene Anzahl von Prozessoren zu finden. Insbesondere bei irregulären Applikationen hängt jedoch die Menge der auszuführenden Tasks oft von der Eingabe und nur zur Laufzeit bekannten Parametern ab, wodurch statisches Scheduling mangels vollständiger Kenntnis des Graphen nicht möglich ist. Dynamisches Taskscheduling sorgt dafür, dass möglichst zu jedem Zeitpunkt jede Recheneinheit eine ausführungsbereite Task abarbeitet, sofern dies die Menge der verfügbaren Tasks erlaubt. Somit muss eine Laufzeitumgebung einen neuen Task einem Prozessor zuweisen, sobald dieser mit der vorherigen Task fertig ist. Für die Entscheidungsfindung kann dabei unterschiedlich viel Aufwand betrieben werden, um eine unter gegebenen Parametern möglichst gute Auswahl aus der aktuellen Menge von ausführungsbereiten Tasks zu treffen. Dieser Aufwand stellt einen Overhead dar und sollte möglichst gering sein. Insbesondere sollte der Aufwand zur Bestimmung des als nächstes auszuführenden Tasks deutlich unter der eigentlichen Tasklaufzeit bleiben.

Die *Taskgranularität* beschreibt in diesem Kontext die Größe des mit einem Task verbundenen Rechenaufwands. Dies kann die Summe der Ausführungszeiten aller Rechenoperationen sein. Im Folgenden werden jedoch zusätzliche Wartezeiten innerhalb einer Task mit berücksichtigt, sodass die Differenz der Start- und Endzeit einer Task dessen Granularität darstellt.

Im Kontext dieser Arbeit beschreiben *feingranulare* Tasks meist Tasks mit Laufzeiten von weniger als einer Sekunde, während *grobgranulare* Tasks wesentlich länger laufen (mehrere Minuten).

Der Schwerpunkt dieser Arbeit liegt in der Untersuchung feingranularer Applikationen, da diese besonderes Potenzial für die Ausnutzung von Rechnersystemen mit einer großen Anzahl von Prozessoren bzw. Cores haben. Allerdings stellen Tasks mit kurzer Laufzeit auch besondere Anforderungen an das entsprechende Laufzeitsystem, da der Overhead für die Taskverwaltung einen größeren Anteil an der Gesamtlaufzeit haben kann und entsprechend besonders effiziente Taskverwaltungsmethoden eingesetzt werden müssen.

Taskzerlegung. Um ein Programm parallel taskbasiert ausführen zu können, müssen Teilaufgaben entsprechend als Task formuliert werden. Dazu werden applikationsabhängig die Programmteile identifiziert, die eine eigenständige Berechnungseinheit darstellen und parallel mit anderen Tasks ausgeführt werden können. Für eine möglichst hohe Parallelität sollte das Programm in möglichst kleine Tasks zerlegt werden. Mit Hilfe vieler kleiner Tasks kann ein gegebener Parallelrechner möglicherweise besser ausgenutzt werden als mit wenigeren, aber größeren Tasks. Allerdings muss aufgrund des Taskverwaltungsoverheads ein Kompromiss bei der Taskgröße gefunden werden.

Die Tasks können einzelne Iterationen einer Schleife darstellen oder Iterationen mehrerer verschachtelter Schleifen. Rekursive Funktionsaufrufe sind ebenfalls mögliche Kandidaten für eine Zerlegung in Tasks. Generell können Funktionen oder Teile von Funktionen als Tasks ausgelegt werden. Da eine Abschätzung der tatsächlichen Laufzeit eines Tasks oft nicht einfach ist, bietet das entwickelte Framework Möglichkeiten, die Taskgröße in einem Profiling-

Schritt zu analysieren. Diese sind in Kapitel 7 beschrieben. Mit Hilfe dieser Informationen kann gegebenenfalls das betrachtete Programm angepasst werden, da für unterschiedliche Architekturen eventuell verschiedene Taskgrößen zu den jeweils kürzesten Gesamtlaufzeiten führen.

Taskabhängigkeiten. Die Ausführung der Tasks kann je nach Applikation gewissen Beschränkungen unterliegen. Manche Tasks können in beliebiger Reihenfolge ausgeführt werden, andere dürfen erst nach Beendigung anderer Tasks ausgeführt werden, um eine korrekte Berechnung zu garantieren. Beispielsweise dürfen in phasenbasierten Berechnungen keine Tasks einer Folgephase berechnet werden, bevor nicht alle Tasks der vorangegangenen Phase abgeschlossen wurden.

Im Allgemeinen sind diese Beschränkungen durch Taskabhängigkeiten erfasst. Dabei können die Abhängigkeiten explizit berücksichtigt werden, wodurch nicht unerhebliche Anforderungen an das Laufzeitsystem gestellt werden. Alle gespeicherten Tasks müssen zwischen ausführungsbereiten und wartenden Tasks unterschieden werden. Entsprechend müssen bei Beendigung eines Tasks davon abhängige Tasks untersucht und gegebenenfalls ausführungsbereit gemacht werden.

Ein alternativer Ansatz, wie er in dieser Arbeit betrachtet wird, ist die Berücksichtigung der Abhängigkeiten durch die Erzeugungsstruktur in der Applikation. Tasks werden erst dann erzeugt, wenn die Abhängigkeiten erfüllt sind. Dadurch können besonders feingranulare Tasks effizient gehandhabt werden. Beispielsweise können bei phasenbasierten Berechnungen erst alle vorhandenen Tasks verarbeitet werden, bevor die Tasks der Nachfolgephase erzeugt werden. Eventuell können dadurch sogar zusätzliche explizite Synchronisationsoperationen innerhalb der Applikation eingespart werden. Auch bei in mehrere Tasks zerteilte Berechnungen eines zusammenhängenden Problems können Folgeberechnungen derart modelliert werden, dass der Folgetask erst vom zuletzt ausgeführten Task der Teilberechnungen erzeugt wird. Diese Methode wurde beispielsweise in einer der in dieser Arbeit betrachteten Applikationen erfolgreich eingesetzt (vgl. Radiosity-Applikation in Kapitel 3.4).

Andere Frameworks wie Cilk [107] können Abhängigkeiten durch ein sogenanntes „Fork-Join"-Modell realisieren. Parallel abzuarbeitende Bereiche werden mit einem „fork"-Befehl erzeugt (auch als „spawn" bekannt) und abgearbeitet. Mit einem expliziten „join"- oder „wait"-Befehl wird dann auf das Ende der so erzeugten Tasks gewartet. Dies erlaubt beispielsweise eine recht einfache Implementierung von „Divide and Conquer"-Algorithmen. Ein Problem dieses Ansatzes ist jedoch, dass eine Fortsetzung nach dem „join"-Befehl innerhalb des gleichen Threads in der korrekten Stackhierarchie erfolgen muss, da der lokale Zustand der Funktion vor dem „join" auch danach erhalten bleiben muss. Während der Wartezeit ist also ein Thread blockiert. In TBB [111] kann der wartende Thread stattdessen einen der erzeugten Tasks ausführen, allerdings ist oft nicht bekannt, wie lange die Berechnungen dauern. Der Thread muss also eventuell dennoch auf die restlichen Tasks warten. Man kann die Wartezeit überbrücken, indem beliebige andere Tasks ausgeführt werden. Allerdings besteht dann die Gefahr, dass die Fortsetzung nach dem „join"-Befehl verzögert wird, wenn die erzeugten Tasks fertig sind, aber der Hauptthread noch einen anderen Task berechnet. Außerdem können durch diese Methode Deadlock-Situationen entstehen, wenn nach dem "join" eine Ressource freigegeben werden müsste, der verantwortliche Thread aber

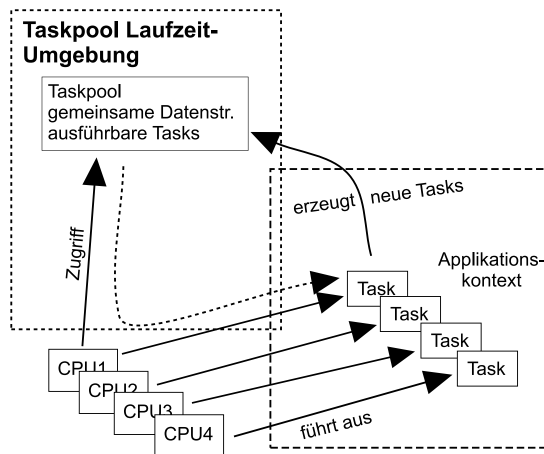


Abbildung 2.1: Schema einer parallelen Programmausführung mit einem Taskpool.

einen Task zur Überbrückung der Wartezeit ausführt, der wiederum auf eine andere Ressource wartet. Werden die Operationen nach dem "join" jedoch als eigener Task spezifiziert, kann ein beliebiger anderer Thread währenddessen die Abarbeitung übernehmen.

Taskpool. Die Speicherung und Ausführung der Tasks einer Applikation übernimmt der Taskpool. In diesem werden ausführungsbereite Tasks gespeichert und an die zugreifenden Threads bei Bedarf verteilt. Der Taskpool stellt ein abstraktes Interface dar, der die eigentliche Speicherung versteckt. Unterschiedliche Implementierungen sind also möglich, um auf Gegebenheiten der Applikation oder der Hardware reagieren zu können. Abbildung 2.1 zeigt den schematischen Aufbau. Der globale Taskpool (dem nicht unbedingt eine zentrale Implementierung zu Grunde liegen muss) speichert die Tasks in einer geeigneten Datenstruktur und wird parallel von den Threads zugegriffen. Der Taskpool entscheidet über die Zuteilung der ausführungsbereiten Tasks entsprechend der jeweiligen Implementierung. Die ausgewählten Tasks werden von den Threads ausgeführt und können neue Tasks erzeugen, die wiederum in dem Taskpool gespeichert werden. Die Entnahme von Tasks zur Ausführung wird auf diese Weise solange wiederholt, bis alle Tasks abgearbeitet wurden.

Die interne Speicherung der Tasks kann auf verschiedene Arten erfolgen. Wesentlicher Unterschied ist dabei die Verteilung der Daten auf die konkurrierenden Threads:

- **zentrale Speicherung:** Der Taskpool nutzt eine zentrale Datenstruktur, um die ausführungsbereiten Tasks zu speichern. Alle Threads greifen dabei konkurrierend auf die Datenstruktur zu. Die Threads müssen entsprechend synchronisiert werden, um eine korrekte Ausführung zu garantieren. Da alle Tasks an einer einzigen Stelle gespeichert sind, sind Lastbalancierungsverfahren nicht nötig. Jeder Thread kann unmittelbar ohne zusätzliche Operationen einen ausführungsbereiten Task entnehmen, sofern solche verfügbar sind. Der Nachteil ist der hohe Synchronisationsaufwand durch den häufigen Zugriff vieler Threads.

- **verteilte Speicherung:** Eine Alternative zu einer zentralen Speicherung besteht in

der Nutzung mehrerer Datenstrukturen, die auf Gruppen von Threads verteilt werden. Dadurch verteilen sich auch die Zugriffe, sodass sich die Last für jede einzelne Datenstruktur reduziert. Allgemein kann eine beliebige Anzahl von Datenstrukturen auf die vorhandenen Threads verteilt werden, also eine $m \times n$-Verteilung. Je mehr Datenstrukturen verwendet werden, desto geringer ist die Wahrscheinlichkeit eines Zugriffskonflikts auf eine einzelne Datenstruktur. Um alle Threads zu jedem Zeitpunkt mit Berechnungen auszulasten, müssen alle verfügbaren Tasks bei der Zuteilung der Tasks betrachtet werden. Entsprechend müssen bei der Benutzung mehrerer Datenstrukturen auch gegebenenfalls mehrere Datenstrukturen nach Tasks durchsucht werden.

- **hierarchische Speicherung:** Als Hybridvariante ist auch ein hierarchischer Ansatz möglich, bei dem auf verschiedenen Stufen unterschiedlich viele Datenstrukturen einer jeweils unterschiedlich großen Teilgruppe von Threads zugewiesen werden. An oberster Hierarchiestufe steht eine zentrale Datenstruktur. Die Hierarchie kann sich beispielsweise an der Speicherhierarchie der jeweiligen Maschine orientieren oder applikationsspezifische Unterteilungen vornehmen. Für die Verteilung der Tasks sind jedoch komplexere Entscheidungen nötig, beispielsweise, in welche Hierarchiestufe neue Tasks platziert werden bzw. wie Umverteilungen zwecks Lastbalancierung durchgeführt werden.

2.2 Aufbau des KOALA-Frameworks

In dieser Arbeit wird die komponentenbasierte adaptive Laufzeitumgebung für taskbasierte Applikationen „KOALA“ entwickelt, die abstrakte Benutzerschnittstellen zur Programmierung irregulärer Applikationen bereitstellt.

Das Framework zur taskbasierten Programmausführung ist in mehrere Schichten unterteilt, die wiederum verschiedene Komponenten enthalten. Abbildung 2.2 zeigt eine schematische Darstellung. Die unterste Schicht beinhaltet Komponenten für Basisfunktionalitäten, die von allen darüber liegenden Komponenten genutzt werden können. Dies umfasst einen Speichermanager (Abschnitt 2.3) zur Speicheranforderung und -freigabe. Die Komponente ist speziell für die effiziente Verwaltung von Speicherobjekten optimiert, die von darüber liegenden Schichten häufig benötigt werden, aber auch in der eigentlichen Applikation Verwendung finden. Ein zweiter Baustein stellt die Lock-Komponente (Abschnitt 2.4) dar, die ein Interface bereitstellt, um wechselseitigen Ausschluss z. B. bei der Benutzung gemeinsamer Datenstrukturen zu ermöglichen. Je nach System und Programmiererwunsch kann zwischen verschiedenen Implementierungen gewählt werden. Neben plattformunabhängigen Lock-Implementierungen für hohe Portierbarkeit existieren auch Hardware-abhängige Implementierungen für hohe Effizienz.

Die nächsthöhere Schicht beinhaltet das Taskpool-Backend (Abschnitt 2.6). Diese Komponente beinhaltet die Lastbalancierungsmechanismen mit Funktionen zum Ablegen und Entnehmen von Tasks. Die tatsächliche Realisierung ist in der Komponente verborgen. Dadurch können verschiedene Lastbalancierungsalgorithmen getestet und eingesetzt werden, ohne dass Änderungen an einer der anderen Komponenten oder der Applikation nötig

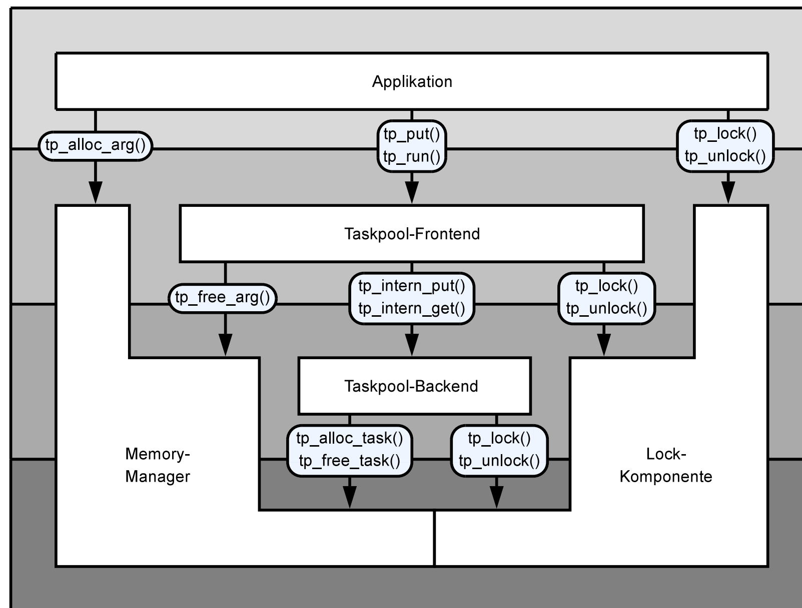


Abbildung 2.2: Schematischer Aufbau des KOALA-Frameworks.

wären. Die jeweilige Implementierung benutzt die Speichermanager-Komponente, um Datenblöcke zur Speicherung der Tasks zu erhalten und die Lock-Komponente, um Zugriffe auf Datenstrukturen anderer Threads innerhalb der eigenen Komponente zu schützen.

Das Taskpool-Frontend (Abschnitt 2.7) stellt die darüber liegende Schicht dar. Die wesentliche Aufgabe besteht in der Bereitstellung eines Interfaces für die oberste Applikationsschicht. Dies umfasst eine Funktion zur Erzeugung und Übergabe neuer Tasks, die entsprechend der tatsächlichen Implementierung dieser Komponente an das darunterliegende Backend weitergegeben werden. Außerdem wird eine Funktion bereitgestellt, um erzeugte Threads an der Taskabarbeitung teilnehmen zu lassen. Zusätzlich kann diese Schicht Statistik- und Debugging-Aufgaben übernehmen. Transparent für die Applikation und das Backend kann ein Profiling-Mechanismus (Kapitel 7) aktiviert werden, um eventuelle Performance-Probleme analysieren zu können.

Die oberste Schicht beinhaltet die eigentliche taskbasierte Applikation. Nach der Erzeugung von Initialtasks und der Übergabe an das Frontend nehmen die von der Applikation erzeugten Threads an der Taskabarbeitung teil. Während der Berechnungen können beliebig weitere Tasks an das Frontend übergeben werden. Dazu greift die Applikation auf die Speichermanager-Komponente zu, um die Eingabedaten der einzelnen Tasks in entsprechenden Datenblöcken zu speichern und zu übergeben. Für wechselseitigen Ausschluss bei dem Zugriff auf gemeinsame Daten innerhalb der Applikation steht bei Bedarf die Lock-Komponente zur Verfügung, die je nach Konfiguration geeignete effiziente Synchronisationsoperationen bereitstellt.

Konfiguration zur Compile-Zeit. Die einzelnen Komponenten sind orthogonal, können also beliebig ausgetauscht und miteinander kombiniert werden. Zur compile-Zeit werden

die zu nutzenden Komponenten ausgewählt und zusammen mit der eigentlichen Applikation zu einem Programm zusammengesetzt. Die Auswahl aus den verfügbaren Komponenten erfolgt dabei automatisch, sodass für jede Komponente eine auf der Zielplattform verfügbare Implementierung benutzt wird. Somit wird ohne weiteres Zutun des Nutzers hohe Portierbarkeit erreicht.

Darüber hinaus können bei Bedarf einzelne Komponenten durch andere Implementierungen ausgetauscht werden, um zum Beispiel verbesserte Synchronisationsoperationen spezieller Prozessoren nutzen zu können.

Das Build-System basiert auf *GNU Autotools*, einer Sammlung frei verfügbarer Programme zur Konfiguration und Übersetzung von Software-Projekten. Im Speziellen bedeutet dies, dass sich eine Standardkonfiguration des Frameworks mittels

```
./configure && make
```

konfigurieren und übersetzen lässt. Ein Aufruf von *./configure --help* zeigt eine Übersicht der verfügbaren Optionen. So lassen sich mit zusätzlichen Optionen wie *--enable-x* oder *--with-y* einzelne Komponenten mit den entsprechenden Bezeichnungen x bzw. y aktivieren.

Laufzeitinitialisierung. Vor der ersten Benutzung eines Framework-Interfaces muss die Taskpool-Laufzeitumgebung initialisiert werden. Dazu dient der Aufruf der Funktion *initTPFramework*:

```
int initTPFramework( int *argc, char ***argv );
```

Als Argumente dieser Funktion werden die Adressen der beiden Argumente der Programmeintrittsfunktion *main()* übergeben. Diese Funktion filtert spezielle Argumente aus der Liste aller Argumente heraus, die für das Framework von Bedeutung sind. Da die entsprechenden Argumente nicht an die Applikation weitergereicht werden, können Argumente für das Framework und die Applikation beliebig vermischt werden, es ist also keine spezielle Behandlung notwendig. Sofern nicht anders angegeben, wird jeweils nur das erste Vorkommen herausgenommen, sodass auch gleichlautende Argumente durch doppelte Angabe möglich sind.

Folgende Argumente dienen der Steuerung des KOALA-Frameworks:

-V, --version Ausgabe der Version des Taskpool-Frameworks.

-v, --verbose Es werden zusätzliche Ausgaben über die Parameter des Frameworks erzeugt.

-l, --log <logfile> Wenn eine Komponente dies benötigt, kann mit dieser Option eine Ausgabedatei angegeben werden, um z. B. trace-Informationen abzuspeichern.

--init-p <threads> Einige Komponenten müssen für das Zielsystem kalibriert werden. Mit dieser Option wird die Anzahl der dafür benutzten Threads angegeben. Das genaue Vorgehen wird in Kapitel 5.4 beschrieben.

Nach dieser globalen Initialisierung des KOALA-Frameworks erfolgt die Erzeugung der nötigen Datenstrukturen für die spezielle Problemstellung mit *tp_create*:

```
int tp_create( int threads, int arg_size );
```

Das Argument *threads* gibt an, wie viele Threads an der Abarbeitung der erzeugten Tasks teilnehmen werden. Da dies mehrfach mit unterschiedlichen Parametern durchgeführt werden kann, bleibt die Erzeugung der Threads der Applikation überlassen. Auf Basis dieser Parameter initialisieren die Komponenten entsprechend ihre internen Datenstrukturen. Die Threads werden intern von 0 bis $threads - 1$ durchnummeriert. Das zweite Argument *arg_size* gibt die maximale Größe eines Taskarguments an, mit dem für jeden Task die Eingabedaten beschrieben werden. Dies wird in Kapitel 2.3 näher erläutert, kann im Wesentlichen aber jeden beliebigen Wert annehmen. Der Rückgabewert ist *TP_OK* im Falle eines Erfolges und *TP_ERROR* im Fehlerfall.

Nach erfolgreicher Initialisierung kann die Taskerzeugung und Taskabarbeitung erfolgen, diese Schritte sind in Kapitel 2.7 beschrieben.

Die Datenstrukturen können mit *tp_finalize* wieder freigegeben werden:

```
void tp_finalize();
```

Anschließend kann das KOALA-Framework für andere Problemstellungen vorbereitet werden, indem *tp_create* erneut mit eventuell anderen Parametern aufgerufen wird.

2.3 Speichermanager

Der Speichermanager sorgt für eine effiziente Speicherverwaltung, die für die speziellen Anforderungen taskbasierter Applikationen und verschiedener Taskpool-Implementierungen optimiert ist. Dabei ist der Speichermanager für zwei Aufgabengebiete verantwortlich. Er behandelt Speicheranfragen der Applikation und getrennt davon Speicheranfragen der Taskpool-Komponenten. Diese beiden Komponenten benötigen für die taskbasierte Programmabarbeitung eine effiziente Verwaltung von Speicherblöcken, um insbesondere feingranulare Tasks verwalten zu können. Durch Einschränkung der möglichen Allokationsgrößen kann der Speichermanager effizienter Speicherblöcke bereitstellen als es das Betriebssystem mit dessen allgemeiner Speicherverwaltung könnte. Er ist spezialisiert auf die Behandlung von Speicherblöcken fester Größe, die sehr häufig angefordert und freigegeben werden. Er wird dann eingesetzt, wenn die Systemaufrufe *malloc* und *free* zu viel Overhead durch Aufrufe der entsprechenden Bibliotheksfunktionen und des Betriebssystems erzeugen würden.

2.3.1 Schnittstellen für die Applikation

Für die Applikationsschicht stellt der Speichermanager ein Interface bereit, mit dem effizient Speicherblöcke mit festgelegter Größe allokiert werden können. Einsatzzweck sind die Taskargumente, mit denen die Eingabedaten eines neu erzeugten Tasks festgelegt werden. Da der Umfang der Daten allein von der Art der Applikation abhängt, muss der Speichermanager flexibel auf die Bedürfnisse reagieren können. Dazu gibt die Applikation mit Hilfe des zweiten Arguments der Funktion *tp_create* (vgl. Kapitel 2.2) an, wie viel Bytes maximal zur Beschreibung eines Taskarguments benötigt werden.

Das Taskargument kann z. B. bei einer Ray-Tracing-Applikation die Koordinaten eines zu berechnenden Pixels sein, also 2×4 Bytes (in Abhängigkeit des Systems). Andere Applikationen benötigen eventuell mehr Speicher, bei anderen wiederum reicht ein einfacher Zeiger auf schon existierende statische Daten.

Der Speichermanager garantiert, dass eine Speicheranforderung einen Zeiger auf einen Datenblock mindestens der angegebenen Größe zurückgibt. Die interne Handhabung der entsprechenden Speicherblöcke obliegt vollständig der jeweiligen Implementierung.

Die Funktionen zur Anforderung und Freigabe solcher Speicherblöcke lauten ähnlich wie *malloc* und *free* aus der C-Standardbibliothek wie folgt:

```
void *tp_alloc_arg( int ID );
void  tp_free_arg ( int ID, void *argument );
```

Beide Funktionen erwarten als erstes Argument die eindeutige Nummer des aufrufenden Threads ($\text{ID} \in [0, \textit{threads})$). Die Thread-ID wird von einigen Implementierungen benötigt, um die Anzahl von Synchronisationsoperationen bei der Anforderung und Freigabe zu reduzieren.

Die Funktion *tp_alloc_arg* gibt einen Zeiger auf einen Speicherblock zurück, dessen Größe mindestens soviel Bytes umfasst, wie bei *tp_create* als Argument *arg_size* angegeben wurde. Der Inhalt des Speicherblocks wird nicht initialisiert und kann eventuell Daten von Tasks enthalten, die diesen Speicherblock zuvor genutzt haben.

Das Framework wird selbstständig einen als Taskargument übergebenen Block freigeben, wenn die Abarbeitung des zugehörigen Tasks abgeschlossen ist. Der Speicherblock darf daher nicht explizit innerhalb des Applikationscodes freigegeben werden.

Anderenfalls kann eine Freigabe mittels Übergabe des Zeigers als zweites Argument von *tp_free_arg* erfolgen.

2.3.2 Schnittstellen für die Taskpool-Schicht

Der zweite Aufgabenbereich des Speichermanagers umfasst die Verwaltung von häufig verwendeten Speicherblöcken für interne Datenstrukturen des Taskpool-Backends. Das Backend muss die erzeugten Tasks zusammen mit den entsprechenden Taskargumenten in geeigneten Datenstrukturen speichern. Verwendet die ausgewählte Backend-Implementierung dynamische Datenstrukturen, so kann es über den Speichermanager schnell auf passende Speicherblöcke zugreifen. Ziel ist auch hier eine größtmögliche Reduktion teurer Systemaufrufe für Speicherallokationen.

Ähnlich wie für die Taskargumente gibt das Taskpool-Backend an, wie viele Bytes ein Speicherblock umfasst. Dazu wird der Aufruf von *tp_create* an das Taskpool-Backend weitergereicht, welches vor der Initialisierung der eigentlichen Datenstrukturen den Speichermanager mittels folgender Funktionen initialisiert:

```
void tp_init_memory( int threads, int taskstruct_size, int arg_size );
void tp_finalize_memory();
```

Die Argumente *threads* und *arg_size* werden, wie von der Applikation vorgegeben, an den Speichermanager weitergereicht. Das zusätzliche Argument *taskstruct_size* gibt die Größe eines vom Backend benutzten Speicherblocks an. Danach können die Blöcke mit folgenden Funktionen allokiert und freigegeben werden:

```
struct tplock_t {
  ...
};

int tplock_init( tplock_t *l );

inline int tplock_lock   ( tplock_t *l );
inline int tplock_unlock( tplock_t *l );
```

Abbildung 2.3: Schnittstelle der Operationen für wechselseitigen Ausschluss.

```
void *tp_alloc_task( int ID );
void  tp_free_task ( int ID, void *task );
```

Analog zu den Funktionen für die Taskargumente muss eine eindeutige Thread-ID übergeben werden. Die Freigabe des Speichers mittels *tp_free_task* erfolgt jedoch nie implizit, dies liegt also ebenfalls im Verantwortungsbereich des Taskpool-Backends.

Der Speichermanager wird vom Backend aber nur für solche Speicherbereiche verwendet, die während einer Taskerzeugung und -abarbeitung häufig benötigt werden. Andere Speicherblöcke, die eventuell nur einmalig allokiert werden, können regulär mittels *malloc* angefordert werden.

2.4 Lock-Komponente

Als zweite Basiskomponente stellt die Lock-Komponente Funktionen zur Synchronisation mehrerer Threads bereit, die je nach den Gegebenheiten des Systems optimierte Operationen benutzen.

Mit Sperrvariablen, den *Locks*, kann die Ausführung einer bestimmten Anweisungsfolge sequenzialisiert werden (der sogenannte *kritische Bereich*). Dazu fordert ein Thread den zugehörigen Lock an. Wenn sich kein anderer Thread in diesem kritischen Bereich befindet, erhält der anfragende Thread den Lock und kann exklusiv im kritischen Bereich z. B. gemeinsame Datenstrukturen modifizieren. Ist die zugehörige Lock-Variable jedoch durch einen anderen Thread gesperrt, wird der anfragende Thread solange blockiert, bis der Lock freigegeben wurde. Da auch mehrere Threads blockiert sein können, wählt die entsprechende Implementierung der Lock-Variablen einen Thread aus dieser Gruppe aus, der als nächstes den kritischen Bereich betreten darf. Somit ist eine sichere Modifikation gemeinsamer Daten sichergestellt.

2.4.1 Lock-Interface

Die tatsächliche Implementierung der Lock-Komponente muss das in Abbildung 2.3 beschriebene Interface implementieren. Die Schnittstelle stellt den Datentyp *tplock_t* bereit, der für die Speicherung der Lock-spezifischen Daten benutzt wird. Der Inhalt der Struktur hängt jedoch einzig von der gewählten Implementierung ab und wird vom Nutzer auch nicht benötigt.

Die Funktion *tplock_init* initialisiert den übergebenen Lock, danach kann die jeweilige Struktur benutzt werden, um mittels der Funktion *tplock_lock* ein exklusives Zugriffsrecht

```
struct tp_duolock_t {
   ...
};

int tp_dl_init( tp_duolock_t *l );

inline int tp_dl_owner_lock   ( tp_duolock_t *l );
inline int tp_dl_owner_unlock( tp_duolock_t *l );
inline int tp_dl_other_lock   ( tp_duolock_t *l );
inline int tp_dl_other_unlock( tp_duolock_t *l );
```

Abbildung 2.4: Schnittstelle der zweistufigen Lock-Operationen.

zu erhalten. Am Ende des kritischen Bereiches erfolgt die Freigabe des Locks mit der Funktion *tplock_unlock*. Beide Funktionen sind explizit als *inline*-Funktionen deklariert, da für sehr feingranulare Locks ein zusätzlicher Funktionsaufruf zur tatsächlichen Lock-Implementierung einen unnötigen Overhead darstellt.

2.4.2 Duolock-Interface

Die Synchronisationsoperationen sollten auch für eine große Anzahl von Prozessoren effizient funktionieren. Allerdings steigt selbst bei der Benutzung von Hardware-nahen Operationen die Zeit zur Sicherstellung wechselseitigen Ausschlusses mit der Anzahl zugreifender Prozessoren an. Innerhalb der Taskpool-Umgebung werden Lock-Operationen aber oft für den Schutz der Datenstrukturen benutzt, die für jeden Thread einzeln existieren. Beispielsweise werden bei verteilten Taskpools jedem Thread eine eigene Datenstruktur zur Ablage ausführungsbereiter Tasks zugeordnet. Dies hat sich als effizient herausgestellt, da jeder Thread nur auf seine eigene Datenstruktur zugreift und nur im Falle einer leeren Struktur auf andere Threads ausweichen muss.

In diesem Szenario sollte der Eigentümer der Datenstruktur ein Vorrangsrecht erhalten, da seine sämtlichen Zugriffsaktionen darauf basieren. Alle anderen zugreifenden Threads suchen im Kontext der Taskpools nach ausführungsbereiten Task nur dann, wenn sie selbst keine ausführungsbereiten Tasks besitzen. Es ist daher sinnvoll, den Zugriff des Eigentümers der Datenstruktur zu beschleunigen und dafür eine Verlangsamung der anderen zugreifenden Threads zu tolerieren, da die Task-suchenden Threads ohnehin keine neuen Tasks erzeugen können, der eigentliche Eigentümer bei der Bearbeitung der eigenen Tasks eventuell aber schon.

Der sogenannte Duolock benutzt ein zweistufiges Lock-System, um einerseits dem Eigentümer Priorität zu geben und andererseits effizientere Lock-Mechanismen zu erlauben. Abbildung 2.4 zeigt die entsprechende Schnittstelle in der Lock-Komponente. Die zugreifenden Threads werden in zwei Klassen unterteilt. Ein Thread tritt als Eigentümer auf, alle anderen Threads sind in der zweiten Klasse enthalten (*externe Threads*). Die Zuordnung wird dabei nicht fest vorgegeben, sondern vom Programmierer durch die Benutzung der korrespondierenden Funktion vorgegeben. Der Eigentümer der mit dem Lock bedachten Datenstruktur verwendet ausschließlich die beiden Funktionen *tp_dl_owner_lock* zur Lock-Anfrage und *tp_dl_owner_unlock* zur Freigabe. Alle anderen Threads müssen für den Zugriff auf die Datenstruktur die beiden anderen Funktionen *tp_dl_other_lock* und

```
struct tp_barrier_t {
  ...
};

void tp_barrier_init( tp_barrier_t *bar );
void tp_barrier( tp_barrier_t *bar, int threads );

inline void hw_code_barrier();
inline void hw_mem_barrier();
```

Abbildung 2.5: Schnittstelle der Barrier-Operationen.

tp_dl_other_unlock benutzen.

In der inneren Stufe des Duolocks tritt die Synchronisation zwischen den beiden Klassen ein. Der Eigentümer und nur ein Thread aus der Klasse der externen Threads können den Lock der inneren Stufe anfragen. Somit reduziert sich das Synchronisationsproblem auf nur zwei Teilnehmer. Es lassen sich so auch einfachere und eventuell schnellere Synchronisationsmechanismen einsetzen. Beispielsweise hat Dekkers Algorithmus [32] zur Synchronisation von zwei Threads geringe Anforderungen an die Hardware, eignet sich aber nur für zwei Teilnehmer.

Die äußere Duolock-Stufe synchronisiert alle zugreifenden Threads der zweiten Klasse untereinander, bevor einer davon mit dem Eigentümer der Datenstruktur um den inneren Lock konkurrieren kann. Dieser Lock muss also eine beliebige Anzahl von Threads wechselseitig ausschließen können. Dazu wird in der Implementierung auf den normalen Lock *tplock_t* zurückgegriffen, somit steht dafür je nach Konfiguration auch eine effiziente Implementierung zur Verfügung.

Zusammengefasst muss der Eigentümer nur den inneren Lock anfragen, um die Datenstruktur zu modifizieren. Alle anderen Threads müssen zuerst den äußeren Lock anfragen und danach den inneren Lock. Das Ziel ist eine Verringerung der Zugriffszeit für die häufige Situation, in der nur der Eigentümer auf die Datenstruktur zugreift, ein Task-Stealing mit Zugriff auf fremde Datenstrukturen aber eher selten passiert. Für diesen Fall werden die stehlenden Threads erst untereinander sequenzialisiert, bevor jeweils nur einer den Eigentümer durch einen Zugriff eventuell behindert.

2.4.3 Barrier-Interface

Ein drittes Element der Lock-Komponente besteht in einer Barrier-Implementierung. Barrieren werden in parallelen Applikationen verwendet, um die laufenden Threads in der Programmabarbeitung zu unterbrechen, bis alle Threads einen bestimmten Synchronisationspunkt erreicht haben. Gemeinsam wird dann die Programmabarbeitung fortgesetzt. Da nicht alle Thread-Bibliotheken der verschiedenen Systeme eine eigene Barrier-Implementierung mitbringen, stellt die Lock-Komponente eine universelle Implementierung bereit.

Einsatzzweck solcher Barrieren sind häufig phasenbasierte Berechnungen, bei denen die Berechnungen einer Folgephase erst dann starten dürfen, wenn alle Ergebnisse der vorherigen Phase vorhanden sind. Auch iterative Probleme erfordern oft eine Barrier-Synchronisation, wenn eine Iteration erst komplett berechnet worden sein muss, bevor die nächste

Iteration berechnet wird.

Die Schnittstelle, wie in Abbildung 2.5 dargestellt, stellt die Datenstruktur *tp_barrier_t* bereit, welche mit der Funktion *tp_barrier_init* initialisiert werden muss. Anschließend kann die Barriere für eine gegebene Thread-Zahl *threads* mit der Funktion *tp_barrier* benutzt werden. Dazu rufen genau *threads* Threads an dem betrachteten Synchronisationspunkt diese Funktion mit der entsprechenden Barrier-Datenstruktur als Argument auf. Jeder aufrufende Thread wird dann solange blockieren, bis alle Threads die Funktion aufgerufen haben. Somit wird die Synchronisation sichergestellt und alle Threads können daraufhin mit der Programmausführung fortfahren.

Die Barriere kann dabei wiederholt wiederverwendet werden, d. h. ein Thread kann die gleiche Barriere aufrufen (wenn er z. B. die nächste Phase einer Berechnung schon erreicht hat), auch wenn andere Threads zwar aufgeweckt, aber den Programmcode der Barriere noch nicht vollständig verlassen haben.

Eine zweite wichtige Klasse von Barrieren sind die Hardware-unterstützten Barrieren, die mit *hw_code_barrier* und *hw_mem_barrier* ebenfalls in der Lock-Komponente zur Verfügung stehen. Diese Funktionen unterstützen den Programmierer bei Programmstellen, die zwar keinen Lock erfordern, aber ein deterministisches Verhalten insbesondere in Multi-Prozessor-Systemen erfordern.

Die Funktion *hw_code_barrier* sorgt dafür, dass der Compiler keine Anweisungen derart verschiebt, dass eine Anweisung vor der Barriere erst danach oder eine Anweisung hinter der Barriere schon vorher ausgeführt wird. Insbesondere bei der Modifikation paralleler Datenstrukturen müssen eventuell zwei verschiedene Datenstrukturen in einer vorgegeben Reihenfolge modifiziert werden. Der Compiler kann die Abhängigkeit aber nicht erkennen und könnte Anweisungen vertauschen, um eine eventuell schnellere Ausführung zu erzielen.

Dementsprechend ist die Funktion *hw_code_barrier* nur zur Compile-Zeit relevant und erzeugt meist auch keinen zusätzlichen Code. Die zweite Funktion *hw_mem_barrier* dagegen stellt eine Sperre zur Laufzeit dar. Sie garantiert, dass alle Lese- oder Schreibzugriffe vor der Programmstelle mit der Anweisung *hw_mem_barrier* tatsächlich abgeschlossen sind, bevor nachfolgende Lese- oder Schreibzugriffe ausgeführt werden. Die Motivation dafür sind die nicht direkt von der Software kontrollierbaren Optimierungsmechanismen der Prozessoren. *Out-of-Order*-Prozessoren ziehen Operationen in der Prozessor-Pipeline nach Möglichkeit vor, wenn dies der aktuelle Prozessorzustand erlaubt und als vorteilhaft bewertet wird. So kann eine Leseoperation schon ausgeführt werden, bevor eine vorherige Schreiboperation abgeschlossen wurde. Spielt die Reihenfolge aber eine Rolle, was häufig bei wechselseitigem Zugriff mehrerer Threads auf eine Datenstruktur der Fall ist, so kann dieses Verhalten zu *race conditions* führen, also fehlerhaften Programmabläufen. Die Funktion *hw_mem_barrier* stellt die im Programm vorgegebene Reihenfolge sicher, allerdings aller Lade- und Speicheroperationen, es werden also gegebenenfalls mehr Operationen synchronisiert als für eine bestimmte Konfliktsituation nötig.

Wenn kein anderweitiger wechselseitiger Ausschluss sichergestellt ist, können diese beiden Funktionen also erforderlich sein, um parallel Datenstrukturen sicher zu modifizieren.

```
struct timesnap_t {
  ...
};

void timesnap( timesnap_t *val );

double timesnapDiffReal( timesnap_t *val_new,
                         timesnap_t *val_old );
double timesnapDiffUser( timesnap_t *val_new,
                         timesnap_t *val_old );
double timesnapDiffSys  ( timesnap_t *val_new,
                         timesnap_t *val_old );
void   printTimeDiff    ( timesnap_t *val_new,
                         timesnap_t *val_old );
```

Abbildung 2.6: Schnittstelle der Zeitmessungen.

2.5 Unterstützende Tools

Einige Implementierungen von Komponenten benötigen zur effizienten Ausführung weitere Funktionalität, die hier kurz dargestellt wird.

Zeitmessung. Eine Zeitmessung wird häufig für die Analyse und Performance-Bewertung von Applikationen benötigt. Die in Abbildung 2.6 gezeigten Funktionen erlauben die Messungen der real verstrichenen Zeit (*real time*), der verbrauchten CPU-Zeit im Programm (*user time*) und der verbrauchten CPU-Zeit im Betriebssystem (*system time*) abstrahiert von den tatsächlichen Funktionen der verschiedenen Betriebssysteme. Die Funktionsgruppe *timesnapDiff* kann benutzt werden, um die Zeitdifferenz der jeweiligen Klasse als Fließkommazahl mit der Genauigkeit der entsprechenden Bibliotheksfunktionen zu bestimmen. Die Einheit des Messwertes ist Sekunden.

Die Zeitmessung mit *timesnap* hat jedoch eine relativ hohe Ausführungszeit, sodass diese Funktionen für laufzeitkritische Komponenten eventuell nicht benutzt werden können oder zu ungenauen Ergebnissen führen. Für effizientere Zeitmessungen bieten viele Prozessoren mittlerweile Instruktionen an, mit denen ohne Umweg über Bibliotheken oder das Betriebssystem ein Zeitstempel ermittelt werden kann. Abbildung 2.7 zeigt das dafür vorgesehene Interface im KOALA-Framework. Ein 64-Bit-Ganzzahlwert wird als Zähler verwendet und von der Funktion *hw_timesnap* mit einem Hardware-abhängigen Zeitstempel gefüllt. Die vom Framework bereitgestellte Konstante *ITC_CLOCK_RATE* gibt die Auflösung des Zeitzählers an. Die genaue Implementierung und Aussagen über die Genauigkeit werden im entsprechenden Kapitel 5.3.3 behandelt. Als Hilfsfunktion kann die Funktion *hw_time_diff* ähnlich wie für *timesnap* benutzt werden, um die Differenz zweier Zeitstempel zu erhalten. Obwohl das im Wesentlichen nur die Differenz der beiden Ganzzahlwerte ist, werden darüber hinaus Überlaufüberprüfungen durchgeführt, da die Auflösung je nach System eventuell nicht groß genug ist, um ohne Überlauf arbeiten zu können.

Speicherlisten. Bei der Implementierung der Applikationen, aber auch der Framework-Komponenten, werden häufig dynamische Datenstrukturen verwendet, die Speicherallokationen erfordern. Ähnlich wie für Taskpool-spezifische Datenstrukturen bietet das in Abbil-

```
uint64_t hw_timesnap();

uint64_t hw_time_diff( const uint64_t t2, const uint64_t t1 );

const uint64_t ITC_CLOCK_RATE;
```

Abbildung 2.7: Schnittstelle der Hardware-unterstützten Zeitmessungen.

```
struct memlist_t {
  ...
};

void  memlist_init ( memlist_t *memlist,
                     int threads, int item_size, int pad_size,
                     memlist_item_init_f init_function );
void  memlist_fini ( memlist_t *memlist );

void *memlist_alloc( memlist_t *memlist, int ID );
void  memlist_free ( memlist_t *memlist, int ID, void *argument );
```

Abbildung 2.8: Schnittstelle der parallelen Speicherliste.

dung 2.8 gezeigte Interface eine allgemeine Möglichkeit, effizient Speicherblöcke anzufordern und freizugeben. Die Speicherliste kann für beliebige Speicherblöcke fester Größe benutzt werden, wobei für jeden unterschiedlichen Datentyp eine eigene Liste mit *memlist_init* initialisiert werden kann. Die Struktur *memlist_t* speichert alle relevanten Informationen der jeweiligen Listen-Instanz, der eigentliche Inhalt ist dabei für den Programmierer unerheblich und bleibt verborgen.

Die Initialisierungsfunktion *memlist_init* erhält neben dieser Struktur die Anzahl von zugreifenden Threads *threads* als Argument, die bei der späteren Nutzung von $0 \dots (threads - 1)$ durchnummeriert sein müssen. Des Weiteren gibt das Argument *item_size* die Größe eines Elements in Bytes an.

Die Speicherlisten-Implementierung kann zusätzliche Bytes zwischen den Datenelementen einfügen, um z. B. die einzelnen Elemente in verschiedenen Cache-Zeilen abspeichern zu können. Das Argument *pad_size* kann benutzt werden, um eine spezifische Größe anzugeben, also z. B. 64 für eine Cache-Zeilengröße von 64 Byte. Ist das Auffüllen nicht nötig bzw. der dafür nötige zusätzliche, aber nicht nutzbare Speicherplatz nicht erwünscht, kann mit einem Wert von 1 diese Funktionalität deaktiviert werden. Der häufigere Fall ist jedoch, dass der tatsächliche Wert dem Programmierer nicht bekannt ist. Bei der Übergabe eines negativen Wertes wird in diesem Fall der vom Framework bereitgestellte Wert übernommen, der durch Vorgabe oder Kalibrierung bestimmt wurde (vgl. Abschnitt 5.4).

Das letzte Argument ist ein Funktionszeiger auf eine Funktion zur Initialisierung eines Speicherelements. Für jedes neu allokierte Speicherelement wird diese Funktion mit der Thread-ID und dem Speicherblock als Argument aufgerufen, um gegebenenfalls programmspezifische Initialisierungen durchführen zu können. Später freigegebene und wiederverwendete Elemente werden jedoch nicht erneut initialisiert. Eine Applikation kann dadurch sicherstellen, dass Speicherelemente nach der Allokation mit *memlist_alloc* mit

```
int  tp_intern_create  ( int threads, int arg_size );
void tp_intern_finalize();

typedef enum {
    TP_NEW_TASK,
    TP_NO_NEW_TASK
} tp_put_return_t;

tp_put_return_t tp_intern_init( int ID, tp_function_t function,
                                void *argument );
tp_put_return_t tp_intern_put ( int ID, tp_function_t function,
                                void *argument );

typedef enum {
    TP_NO_TASK,
    TP_GOT_TASK,
    TP_GOT_TASK_WITH_NEW_ONES
} tp_get_return_t;

tp_get_return_t tp_intern_get( int ID, tp_function_t *function,
                               void **argument );
```

Abbildung 2.9: Schnittstelle des Taskpool-Backends.

bestimmten Werten initialisiert sind. Wird diese Funktionalität nicht benötigt, so kann als Funktionszeiger auch *NULL* übergeben werden.

Die eigentliche Allokation eines Speicherelements erfolgt mit *memlist_alloc* unter Übergabe der entsprechenden Speicherliste und der Thread-ID. Eine Freigabe erfolgt mit *memlist_free* mit Angabe des freizugebenden Speicherelements. Da bei einer Freigabe ein Zwischenspeichern der Elemente für spätere Wiederverwendung erfolgt, sollten die Speicherlisten komplett mit *memlist_fini* freigegeben werden, wenn sie nicht mehr benötigt werden.

2.6 Backend

Die Backend-Schicht des KOALA-Frameworks ist für die eigentliche Taskverwaltung zuständig und stellt den eigentlichen Taskpool bereit. Sie speichert neu verfügbare Tasks geeignet ab und gibt auf Anfrage ausführbare Tasks zurück. Lastbalancierungsverfahren wie Task-Stealing werden gegebenenfalls durchgeführt. Die tatsächliche Implementierung kann beliebig ausgetauscht werden und ist für den Applikationsprogrammierer nicht ersichtlich. Er benutzt lediglich die entsprechenden Frontend-Funktionen.

2.6.1 Interface

Eine Taskpool-Implementierung muss das in Abbildung 2.9 dargestellte Interface implementieren, um als Komponente genutzt werden zu können.

Für die Initialisierung der Komponente durch das Frontend wird die Funktion *tp_intern_create* von dieser aufgerufen. Die beiden Argumente geben die Anzahl der teilnehmenden Threads und die Größe der Taskargument-Datenstruktur an, wie sie von der Applikation vorgegeben wird. Die Backend-Implementierung muss dann die eigene Datenstruktur geeignet initialisieren und den Speichermanager der darunterliegenden Schicht durch Aufruf

von *tp_init_memory* initialisieren. Wenn der Taskpool nicht mehr benötigt wird, erfolgt der Aufruf von *tp_intern_finalize*, um die allokierten Datenstrukturen wieder freizugeben und den Speichermanager ebenfalls zu beenden.

Wenn die Applikation neue Tasks erzeugt, werden diese von dem Frontend an die Taskpool-Implementierung im Backend weitergereicht, indem die Funktion *tp_intern_put* aufgerufen wird. Neben der Thread-ID des erzeugenden Threads erwartet die Funktion einen Funktionszeiger, der die abzuarbeitende Funktion des Tasks beschreibt und ein entsprechendes Taskargument, das der Taskfunktion bei ihrer Ausführung übergeben wird.

Für die Initialisierungsphase steht die Funktion *tp_intern_init* bereit, um initiale Tasks in den Taskpool einzufügen. Dabei muss die Thread-ID nicht mehr mit dem aufrufenden Thread übereinstimmen und kann verwendet werden, um Tasks in den lokalen Taskpools anderer Threads abzuspeichern. Je nach Taskpool-Implementierung kann eine initiale Verteilung Auswirkungen auf die Geschwindigkeit haben. Bei der Übergabe einer negativen Thread-ID an *tp_intern_init* werden die Tasks der Reihe nach mit aufsteigenden Thread-IDs in „round-robin"-Ordnung in den Taskpool eingefügt.

Der Rückgabewert der beiden Einfügefunktionen kann die symbolischen Werte *TP_NEW_TASK* oder *TP_NO_NEW_TASK* annehmen, um zu kennzeichnen, ob das Einfügen des neu erzeugten Tasks tatsächlich zu einer globalen Verfügbarkeit neuer Tasks geführt hat. Eine Taskpool-Implementierung kann beispielsweise private Datenstrukturen vorsehen, in denen eine gewisse Menge von Tasks gespeichert wird, welche nicht für Task-Stealing anderer Prozessoren zur Verfügung stehen. Da aber das Frontend wartende Threads eventuell aufwecken muss, um ihnen verfügbare Tasks zur Abarbeitung zuweisen zu können, wird dieser Rückgabewert vom Frontend entsprechend ausgewertet.

Die letzte Funktion *tp_intern_get* wird bei der Taskabarbeitung vom Frontend verwendet, um einen Task für einen Thread mit gegebener ID aus dem Taskpool zu entnehmen. Sofern verfügbar, wird der Funktionszeiger und das Taskargument in den beiden entsprechenden Argumenten zur Ausführung abgelegt. Der Rückgabewert der Funktion gibt dabei an, ob ein Task entnommen wurde (*TP_GOT_TASK*) und ob eventuell neue Tasks verfügbar sind (*TP_GOT_TASK_WITH_NEW_ONES*). Dies kann auftreten, wenn die Taskpool-Implementierung entschieden hat, Tasks aus privaten Datenstrukturen für andere Threads zugänglich zu machen. Konnte kein Task entnommen werden, wird *TP_NO_TASK* zurückgegeben.

2.7 Frontend

Das Framework-Frontend stellt die eigentliche Schnittstelle zwischen Applikation und Taskpool dar. Die Aufgabengebiete sind die korrekte Initialisierung der einzelnen Komponenten, das Weiterleiten neuer erzeugter Tasks an das Backend und die eigentliche Taskabarbeitung.

2.7.1 Interface

Das Frontend kann ebenfalls als Komponente durch eine beliebige Implementierung des in Abbildung 2.10 dargestellten Interfaces ausgetauscht werden.

Wie schon in Kapitel 2.2 dargestellt, werden die Funktionen *initTPFramework*, *tp_create* und *tp_finalize* zur Erzeugung und Freigabe der gesamten Framework-Struktur benutzt. Die

```
int  initTPFramework( int *argc, char ***argv );
int  tp_create  ( int threads, int arg_size );
void tp_finalize();

typedef void (*tp_function_t)(int, void *);
int tp_init( int ID, tp_function_t function, void *argument );
int tp_put ( int ID, tp_function_t function, void *argument );

int tp_run ( int ID );
```

Abbildung 2.10: Schnittstelle des Taskpool-Frontends.

Anzahl der benutzten Threads und die Größe der Taskargument-Datenstruktur wird von der Applikation vorgegeben.

Die wesentliche Aufgabe der Funktion *tp_create* ist die Initialisierung der Frontend-Komponente und die Initialisierung der Backend-Komponente durch Aufruf von *tp_intern_init*.

Die beiden Funktionen *tp_put* und *tp_init* werden von dem Applikationsprogrammierer verwendet, um neue Tasks während der Abarbeitung respektive der Initialisierung einzufügen. Die eigentliche Frontend-Implementierung wird im Wesentlichen die Tasks an die jeweilige Backend-Implementierung durch Aufruf von *tp_intern_put* bzw. *tp_intern_init* weiterreichen, sie kann je nach Bedarf aber zusätzliche Operationen ausführen (z. B. für statistische Zwecke). Insbesondere muss jedoch das Frontend in Abhängigkeit des Rückgabewertes der Backend-Funktionen wartende Threads aufwecken, wenn beispielsweise durch die Einfügeoperation neue Tasks im öffentlich zugänglichen Bereich eines Threads verfügbar sind.

Der wichtigste Teil der Frontend-Implementierung ist die Funktion *tp_run*, die die eigentliche Taskabarbeitung kontrolliert. Nach Initialisierung und der Erzeugung initialer Tasks muss die Applikation jeden teilnehmenden Thread die Funktion *tp_run* mit der entsprechenden Thread-ID aufrufen lassen. Sobald alle Threads die Funktion betreten haben, startet die parallele Abarbeitung der gespeicherten Tasks. Die Funktion kehrt dabei erst dann zurück, wenn es keine weiteren ausführungsbereiten Tasks im gesamtem Taskpool-System gibt und alle anderen Threads ebenfalls keine Tasks mehr bearbeiten. Danach bekommt die Applikation die Kontrolle zurück und kann gegebenenfalls neue Tasks einer folgenden Berechnungsphase erzeugen. Ein erneuter Aufruf von *tp_run* wird diese Tasks dann abarbeiten. Die Funktion stellt dabei eine implizite Barriere dar, sodass Threads schon die nächste Phase betreten dürfen, bevor alle neuen Tasks erzeugt worden sind.

2.7.2 Aufgaben

Das Hauptaufgabengebiet des Frontends ist die parallele Abarbeitung der ausführungsbereiten Tasks. Taskverwaltung und Lastbalancierung wird von der gewählten Backend-Implementierung ausgeführt, sodass das Frontend nur Tasks vom Backend für den jeweiligen Thread anfragen muss. Die strikte Trennung von Ausführung und Verwaltung ermöglicht das Messen verschiedener Performance-Parameter transparent für die Applikation und das Backend. Wenn es nötig erscheint, um z. B. Performance-Probleme zu identifizieren, kann das Frontend eine Implementierung benutzen, die statistische Erhebungen der ausgeführten

Tasks macht. Dies kann einfaches Zählen der ausgeführten Tasks pro Thread über eine Zeitmessung bis hin zur feingranularen Erfassung einzelner Tasks umfassen. Möglich ist auch eine dynamische Erzeugung des Taskgraphen, um nach der Abarbeitung einen genauen Überblick über die taskbasierte Berechnung der Applikation zu erhalten und gegebenenfalls Änderungen in der Applikation vorzunehmen. In Kapitel 7 wird dazu ein Profiling-Frontend beschrieben, mit dem sich die Task-Charakteristik der Applikation analysieren lässt, um Probleme in der parallelen Abarbeitung zu erkennen.

2.8 Beispielapplikation

Um die Programmierung einer taskbasierten Applikation zu verdeutlichen, ist in Abbildung 2.11 ein vereinfachtes Beispiel einer solchen Applikation dargestellt. Die Applikation nutzt als Taskargument eine Struktur zur Speicherung eines einzelnen Integer-Wertes. Entsprechend wird bei der Initialisierung in Zeile 25-26 neben der gewählten Anzahl von Threads (*THREADS*) die Größe dieser Datenstruktur an *tp_create()* übergeben. Anschließend wird in den Zeilen 28-30 ein Initialtask in den Taskpool gelegt, der als Argument einen gegebenen Startwert (*STARTVALUE*) verwendet. Die eigentliche Taskabarbeitung beginnt in Zeile 32-33 durch Erzeugung der Threads und Ausführung der Funktion *tp_run()*. Sind alle Threads mit der Taskabarbeitung fertig, kann das Framework in Zeile 36 beendet werden.

Die eigentliche Taskfunktion *task1* erhält als Argumente die ID des ausführenden Threads und das jeweilige Argument des auszuführenden Tasks. Im dargestellten Beispiel wird geprüft (Zeile 8), ob der Zahlenwert in der Taskargumentstruktur größer als Null ist. In diesem Fall wird in den Zeilen 9-11 ein neuer Task erzeugt, dessen Argument um Eins kleiner ist als das Argument des aktuellen Tasks. Anschließend werden in Zeile 13 Berechnungen in Abhängigkeit von dem Taskargument durchgeführt. Nach Beendigung dieser Operationen wird ein weiterer Task in den Zeilen 15-17 erzeugt, dessen Argument um Zwei kleiner ist als das aktuelle Argument. Anschließend ist dieser Task vollständig abgearbeitet.

Ein Task dieser Beispielapplikation erzeugt also jeweils zwei neue Tasks mit jeweils unterschiedlichem Argument, falls das Taskargument größer als Null ist.

Die wesentlichen Schritte der Implementierung einer realen Applikation bestehen also in der Erzeugung geeigneter Initialtasks, die das zu lösende Problem beschreiben. Anschließend kann die Taskabarbeitung mit der entsprechenden Anzahl von Threads gestartet werden. Die auszuführenden Tasks werden vom Framework den Threads zugewiesen. Um einen möglichst hohen Parallelitätsgrad zu erhalten, sollten die Tasks innerhalb der entsprechenden Taskfunktion nach Möglichkeit neue Tasks erzeugen, um beispielsweise Teilprobleme parallel lösen zu können. Sind alle so erzeugten Tasks abgearbeitet, kehrt die Programmausführung in die Hauptfunktion zurück, sodass dort das Resultat der Berechnungen ausgegeben oder geeignet nachverarbeitet werden kann.

```
1  typedef struct { int i; } arg_t;
2
3  void task1( int ID, void *arg )
4  {
5      arg_t *task1_arg  = (arg_t*)arg;
6      arg_t *new_arg;
7
8      if ( task1_arg->i > 0 ) {
9          new_arg = (arg_t*) tp_alloc_arg( ID );
10         new_arg->i = task1_arg - 1;
11         tp_put( ID, task1, new_arg);
12
13         calculate( task1_arg->i );
14
15         new_arg = (arg_t*) tp_alloc_arg( ID );
16         new_arg->i = task1_arg - 2;
17         tp_put( ID, task1, new_arg);
18     }
19 }
20
21 int main( int argc, char **argv )
22 {
23     arg_t *arg;
24
25     initTPFramework( &argc, &argv );
26     tp_create( THREADS, sizeof( arg_t) );
27
28     arg = (arg_t*)tp_alloc_arg( NUMBER );
29     arg->i = STARTVALUE;
30     tp_init( -1, task1, arg );
31
32     createThreads( THREADS - 1, tp_run );
33     tp_run( THREADS - 1 );
34
35     joinThreads();
36     tp_finalize();
37
38     return 0;
39 }
```

Abbildung 2.11: Vereinfachtes Beispiel einer taskbasierten Applikation.

Kapitel 3

Taskbasierte Applikationen

In Kapitel 2.1 wurde der taskbasierte Ansatz zur parallelen Abarbeitung beschrieben. In diesem Kapitel werden nun die für die weitergehenden Analysen und Performance-Betrachtungen benutzten Applikationen und ihre jeweilige Taskstruktur beschrieben. Die betrachteten Applikationen wurden aufgrund ihrer irregulären Ausführungsschemas ausgewählt. Die Taskcharakteristik ist jeweils verschieden und stellt unterschiedliche Anforderungen an die Taskpool-Implementierung, um eine effiziente Ausführung zu realisieren.

3.1 Synthetische Applikation

Um verschiedene Taskpool-Implementierungen vergleichen und den Overhead bestimmen zu können, hat sich eine synthetische Applikation als nützlich erwiesen, wie sie in [57, 69] beschrieben ist. Ziel dieser Applikation ist es, deterministisch Tasks so zu erzeugen, dass die verschiedenen Taskpool-Mechanismen tatsächlich angesprochen werden. Die synthetische Applikation erlaubt die exakte Kontrolle über die Anzahl der erzeugten Tasks und die Granularität der einzelnen Tasks. Damit verschiedene Lastbalancierungsverfahren getestet werden können, muss der Rechenaufwand für verschiedene Threads unterschiedlich groß sein. Außerdem sollen dynamisch neue Tasks erzeugt werden, um die Datenstrukturen zur Speicherung ausführungsbereiter Tasks testen zu können. Während der Abarbeitung werden Berechnungen simuliert, die ohne gemeinsame Variablen und sonstige Speicherzugriffe auskommen, um den Einfluss auf die Analyse der Taskpool-Komponenten nicht mehr als nötig zu beeinflussen. Da keine Synchronisation zwischen den Tasks nötig ist, lassen sich Geschwindigkeits- und Skalierbarkeitseinschränkungen auf die verwendete Taskpool-Implementierung zurückführen und analysieren.

Abbildung 3.1 zeigt den schematischen Aufbau eines Tasks der synthetischen Applikation. Ein Task wird im Wesentlichen durch ein ganzzahliges Argument *arg* beschrieben. Ist dieses größer als Null, werden im Laufe der Abarbeitung zwei neue Tasks erzeugt, je ein Task mit Argument $arg - 2$ (Zeile 4) und ein Task mit Argument $arg - 1$ (Zeile 6). Die Anzahl der so erzeugten Tasks wächst exponentiell mit dem Taskargument entsprechend einer Fibonacci-Folge. Zwischen der Taskerzeugung werden jeweils Berechnungen durch die Funktion *compute* simuliert, deren Aufwand in linearem Zusammenhang zu dessen Argument steht (Zeilen 3, 5 und 7). Ein globaler Parameter f erlaubt die Veränderung des Rechenaufwands, womit auch besonders kleine oder große Tasks (wenig bzw. viel Rechenaufwand) simuliert werden können.

Ist das Taskargument kleiner oder gleich Null, bricht die rekursive Taskerzeugung ab, der entsprechende Task führt nur simulierte Berechnungen durch (Zeile 9) und erzeugt keine

```
1 syn_task( ID, arg ):
2     if arg > 0:
3         compute( 10 * f )
4         create_task( syn_task, arg - 2 )
5         compute( 50 * f )
6         create_task( syn_task, arg - 1 )
7         compute( 100 * f )
8     else:
9         compute( 100 * f )
```

Abbildung 3.1: Pseudo-Code eines Tasks der synthetischen Applikation.

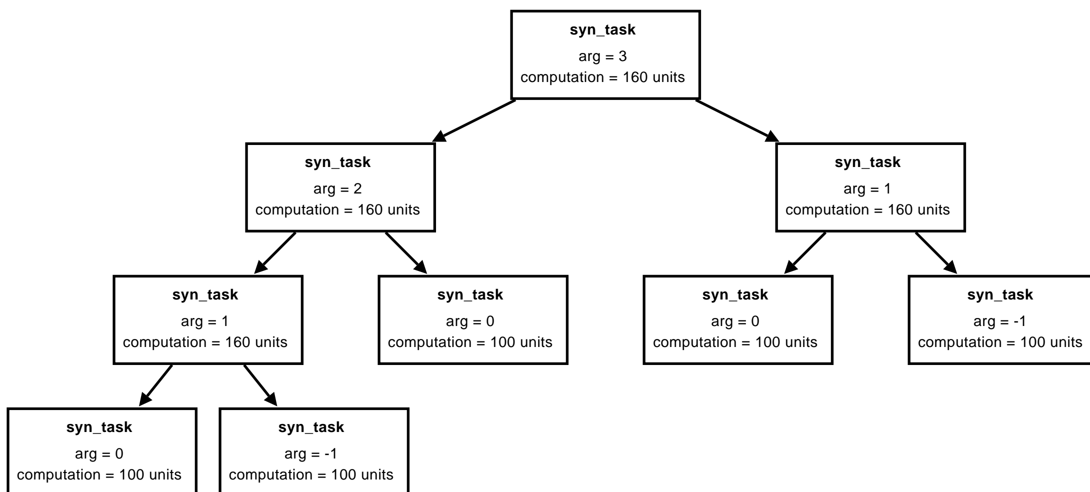


Abbildung 3.2: Beispiel einer Taskerzeugung der synthetischen Applikation für einen Task mit dem Argument 3.

neuen Tasks mehr. Entsprechend lässt sich der mit einer Task mit Argument i verbundenen Berechnungsaufwand S wie folgt rekursiv definieren:

$$S(i) = \begin{cases} 100 \cdot f & \text{for } i \leq 0 \\ 10 \cdot f + S(i-2) + 50 \cdot f + S(i-1) + 100 \cdot f & \text{else} \end{cases}$$

Die Taskerzeugung für einen initialen Task mit dem Argument 3 ist in Abbildung 3.2 beispielhaft dargestellt. Ein Merkmal der synthetischen Applikation ist dabei der Lastunterschied der jeweiligen Tasks. So beschreiben die Tasks im linken Teilbaum der Abbildung $S(2) = 620 \cdot f$ Recheneinheiten, während der rechte Teilbaum nur $S(1) = 360 \cdot f$ Recheneinheiten umfasst. Diese ungleiche Lastverteilung muss durch die Taskpool-Implementierung geeignet ausgeglichen werden, um alle Threads möglichst gleichmäßig mit Berechnungen auszulasten.

Die synthetische Applikation wird durch drei Parameter kontrolliert. Neben der Anzahl zu benutzender Threads steht der Parameter f wie zuvor erwähnt für den mit einer Task verbundenen Rechenaufwand. Für einen Wert $f = 0$ werden „leere“ Tasks simuliert, die also keinerlei Berechnungen ausführen und nur entsprechend des Taskerzeugungsschemas Tasks erstellen. Somit werden die größtmöglichen Belastungen des Taskpools simuliert, da

Argument	Anzahl erzeugter Tasks
20	57.290
25	635.593
30	7.049.122
31	11.405.739
32	18.454.894
33	29.860.667
34	48.315.596
35	78.176.299

Tabelle 3.1: Anzahl insgesamt erzeugter Tasks der synthetischen Applikation für bestimmte Taskargumente.

sehr häufig neue Tasks zur Ausführung gefunden werden müssen und sehr viele Tasks in kurzer Abfolge in den Taskpool eingestellt werden. Die Variation dieses Parameters f erlaubt so eine Untersuchung, welche Taskgröße noch effizient von einer gegebenen Taskpool-Implementierung behandelt werden kann.

Der dritte Parameter t bestimmt die Anzahl initial erzeugter Tasks. Zu Beginn der Applikation wird für jedes $i, 0 \leq i < t$ ein Task mit dem Argument i erzeugt. Anfangs existieren also t Tasks mit jeweils unterschiedlich viel damit verbundenen Rechenaufwand, im Laufe der Abarbeitung wird aber eine wesentlich höhere Anzahl von Tasks erzeugt. Für einen Wert von $t = 35$ werden beispielsweise insgesamt 78.176.299 Tasks erzeugt. Tabelle 3.1 zeigt exemplarisch für einige Argumentwerte die Anzahl der insgesamt erzeugten Tasks.

Die beschriebene synthetische Applikation ist gut geeignet, um die generelle Performance und Skalierbarkeit verschiedener Taskpool-Implementierungen zu untersuchen. Das deterministische Taskerzeugungsschema generiert einen sehr unbalancierten Taskgraphen, sodass verschiedene Lastbalancierungsverfahren verglichen werden können. Durch Modifikation des Parameters f lässt sich der Einfluss der Taskgranularität auf die Performance der Taskpools untersuchen, da sich damit das Verhältnis der Taskberechnungen zu dem Verwaltungsaufwand der notwendigen Taskoperatioen variieren lässt und sich so Schwachstellen erkennen lassen.

3.2 Ray-Tracing

Eine reale Applikation aus dem Bereich der Computergrafik ist das „Ray-Tracing“-Verfahren [118], um aus dreidimensional angeordneten Objekten mit verschiedenen Lichtquellen eine zweidimensionale Darstellung zu errechnen. Für jeden Pixel des zu berechnenden Bildes wird ein Strahl ausgehend von der Kameraposition durch den entsprechenden Punkt der Bildebene in den Objektraum verfolgt. Für jeden Strahl wird ein Schnittpunkt mit einem Objekt berechnet, das die geringste Entfernung zur Strahlenquelle hat. Gibt es ein entsprechendes Objekt, werden neue Strahlen zu allen Lichtquellen verfolgt, um den Einfluss auf die eigentliche Farbe des Pixels durch die Lichtquellen zu bestimmen. Je nach den Oberflächeneigenschaften des getroffenen Objekts wird ein neuer Reflexionsstrahl verfolgt, der wiederum das Objekt sucht, das entsprechend dem Startpunkt die geringste Entfernung

hat. Die Strahlenverfolgung endet, wenn keine weiteren Reflexionen auftreten, der Strahl den Objektraum verlässt oder eine vorgegebene Anzahl von Reflexionen überschritten ist. Die genaue Implementierung basiert auf der Variante von Spach und Pulleyblank [122].

Der mit einem Strahl verbundene Rechenaufwand hängt also stark von der jeweiligen Szene ab und kann nicht im Voraus bestimmt werden. Lastbalancierung ist also notwendig, um die Berechnungen optimal auf die zur Verfügung stehenden Prozessoren zu verteilen. Da die Threads aber nur lesend auf die gemeinsame Szenenbeschreibung zugreifen und zudem in disjunkten Bereichen des Bildes gerechnet wird, ist kein Synchronisationsaufwand nötig, was die parallele Implementierung entsprechend vereinfacht.

In dieser Arbeit wird auf eine modifizierte Implementierung des „Ray-Tracing“-Verfahrens aus der SPLASH2-Programmsammlung [136] zurückgegriffen, bei der einige interne Probleme behoben worden sind. Dies umfasst die Entfernung einer ungenutzten Strahlidentifizierungsnummer, die durch einen Lock geschützt wurde.

Diese Implementierung erzeugt einen Task für je einen Block mehrerer Pixel, der während der Ausführung keine zusätzlichen Tasks erzeugt. Die Blockgröße wird durch die jeweilige Szenendatei vorgeben, beispielsweise 8×8 Pixel pro Block für die Szene „car“. Diese Szene wird in den entsprechenden Experimenten verwendet und enthält 7629 Objekte mit insgesamt 46423 Elementen.

3.3 Volume-Rendering

Das „Volume-Rendering“-Verfahren [98] wird benutzt, um 3D-Modelle graphisch darzustellen. Dazu wird eine dreidimensionale Matrix von Funktionswerten verwendet, die die Modelldichte an den jeweiligen Raumkoordinaten darstellen. Diese Funktionswerte können z. B. Magnetresonanz-Daten (MR) und Computertomographie-Daten (CT) sein. Um ein zweidimensionales Bild zu erhalten, werden Strahlen durch diese Dichtedaten verfolgt. Der Farbwert ergibt sich dann durch Aufsummierung der Dichtedaten an speziell definierten Punkten auf dem Strahl. Je höher die Dichte ist, desto höher ist die Absorption des Strahls. Je nach Dichte des Gewebes kann die Strahlenverfolgung entsprechend eher beendet werden. Eine hierarchische Raumunterteilung [81] wird verwendet, um Gebiete mit geringer Dichte schneller durchlaufen zu können. Außerdem kann eine adaptive Unterteilung [82] angewendet werden, um die Berechnungen zu beschleunigen, wodurch ein höherer Grad von Irregularität auftritt.

Die parallele Implementierung der SPLASH2-Programmsammlung [136] unterteilt das zu erzeugende Bild in gleichgroße Blöcke bestehend aus mehreren Bildkacheln, wobei jede Bildkachel als einzelner Task die eigentliche Berechnungseinheit darstellt. Die Tasks zur Berechnung der Bildkacheln werden am Anfang erzeugt, eine weitere Taskerzeugung zur Berechnungszeit findet nicht statt. Prinzipiell findet nur lesender Zugriff auf die Dichtedaten statt, sodass keine Synchronisation zwischen den Tasks notwendig ist. Allerdings werden bei dem adaptiven Verfahren Teilräume so verarbeitet, dass zuerst nur die Eckpunkte einer Bildkachel berechnet werden und nur bei hohen Dichteunterschieden weitere Unterteilungen stattfinden. Da die Eckpunkte jedoch zu mehreren benachbarten Blöcken gehören, muss ein Task eventuell auf einen anderen Task warten, der gerade einen benötigten Eckpunkt berechnet. Findet dagegen keine Unterteilung statt, werden die dazwischen

liegenden Bildpunkte geeignet interpoliert. Eine genaue Beschreibung ist in [82] zu finden. Für die Untersuchungen in dieser Arbeit werden sowohl die nicht adaptive Variante, die für jeden Bildpunkt auch einen Strahl verfolgt, als auch die adaptive Variante betrachtet, die entsprechend der Szene unterschiedlich viele Strahlen pro Bildkachel verfolgt.

Ähnlich zur „Ray-Tracing"-Applikation hängt also auch hier der Berechnungsaufwand eines Bildpunktes von der eigentlichen Eingabe ab. Zudem ist bei der Benutzung der adaptiven Verfeinerung der Bildkacheln nicht vorhersagbar, welche Berechnungen tatsächlich ausgeführt werden müssen oder welche schon von anderen Tasks berechnet wurden und wiederverwendet werden können. Allerdings ist der zu verarbeitende Datenumfang wesentlich höher als beim „Ray-Tracing"-Verfahren, da keine spezifischen Objekte existieren, sondern eine komplett gefüllte Dichtematrix.

Die für Laufzeitexperimente verwendete Beispielszene „head" ist durch eine Matrix der Größe $256 \times 256 \times 128$ beschrieben.

3.4 Hierarchisches Radiosity

Der globale Beleuchtungsalgorithmus „Hierarchisches Radiosity" [48, 118, 119] berechnet betrachterunabhängig die Lichtverteilung zwischen Objekten in einer dreidimensionalen Szene und berechnet mit diesen Informationen für einen gegebenen Beobachtungspunkt eine zweidimensionale Darstellung. Dazu werden die Objekte einer Szene in n Oberflächenelemente zerlegt, für die der Strahlungsaustausch anhand der Radiosity-Gleichung auf Basis des Energieerhaltungssatzes berechnet wird:

$$B_i = E_i + \rho_i \sum_{j=1}^{n} F_{ij} B_j \qquad , \ i = 1, \dots, n.$$

Die berechneten Radiosity-Werte B_i eines Elements i setzen sich also zusammen aus der jeweiligen Lichtemission E_i des Elements und der Summe der einfallenden Radiosity-Werte B_j der anderen Elemente in der Szene. Der diffuse Reflexionskoeffizient ρ_i gibt als Oberflächeneigenschaft den Anteil der reflektierten Strahlungsmenge an. Der Formfaktor F_{ij} beschreibt jeweils, welcher Anteil der Strahlungsenergie von E_i bei E_j ankommt und hängt von der Lage der beiden Elemente zueinander und dazwischen liegenden anderen Elementen ab. Eine genauere Beschreibung ist z. B. in [108, S.519ff] zu finden.

Eine Besonderheit dieses Algorithmus aus der Sicht einer taskparallelen Abarbeitung ist die dynamische adaptive Objektunterteilung anhand der jeweiligen Strahlungsunterschiede auf der Objektoberfläche, wodurch eine irreguläre Berechnungsstruktur entsteht, die von der Eingabeszene abhängt und nicht vorhersagbar ist. Daher eignet sich diese Applikation besonders für die Untersuchung von Lastbalancierungsverfahren von taskbasierten Applikationen. Als Programmbasis wird die Implementierung in der SPLASH-2-Programmsammlung [136] verwendet.

Im Folgenden soll ein kurzer Überblick über den Aufbau der Applikation gegeben werden, eine genauere Beschreibung lässt sich z. B. in [120, 136] finden. Algorithmus 3.1 zeigt den dreiphasigen Aufbau der Applikation und die für die Berechnungen verwendeten Tasktypen.

Die Eingabeszene beschreibt die Oberfläche der Objekte der Szene durch entsprechende Dreiecke, die neben den Koordinaten im Raum auch Attribute wie Farbe und insbesondere

Algorithmus 3.1 Schematischer Aufbau der hierarchischen Radiosity-Applikation und deren Tasktypen.

```
Phase 1:
  For all Eingabedreiecke P:
    Füge P in BSP-Baum ein;
    Erzeuge „REFINEMENT“-Task für P;

Phase 2:
  Repeat:
    For all Dreiecke P im BSP-Baum:
      Erzeuge „RAY“-Task für P;
    Führe Tasks aus;
  Until Fehlerschranke unterschritten;

Phase 3:
  For all Dreiecke P im BSP-Baum:
    Erzeuge Task AVERAGE(P,average);
  Führe Tasks aus;
  For all Dreiecke P im BSP-Baum:
    Erzeuge Task AVERAGE(P,normalize);
  Führe Tasks aus;

Task REFINEMENT(P):
  For all Dreiecke P’ im BSP-Baum:
    Berechne Formfaktor zwischen P und P’;
    Unterteile, falls nötig, entweder P oder P’ rekursiv;

Task RAY(P):
  For all Interaktionen I von P:
    If Fehler(I) zu groß:
      Unterteile Element(I) und erzeuge neue Interaktionen;
    Else If Interaktion I noch nicht fertig:
      Erzeuge Task VISIBILITY(I);
    Else
      Sammle Energie auf;
      If P ist Blattknoten:
        Schicke Energie zum Vaterknoten;
      Else:
        For each Kind C:
          Erzeuge Task RAY(C);

Task VISIBILITY(P):
  Berechne Sichtbarkeit für Menge von Interaktionen;
  Fahre mit Task RAY(P) fort;

Task AVERAGE( P, mode ):
  If P ist Blattknoten:
    Mittelwert bilden oder Normalisieren;
  Else:
    for each child C:
      Erzeuge Task AVERAGE(C,mode);
```

die ausgestrahlte Strahlungsmenge besitzen. Diese Dreiecke werden in Phase 1 in einem BSP-Baum (Binary Space Partitioning) eingefügt, wobei für jedes Dreieck ein entsprechender „Refinement“-Task erzeugt wird. Dieser Tasktyp unterteilt das Dreieck in Teildreiecke entsprechend ihrer Größe und dem Strahlungsanteil, bis definierte Fehlerschranken erreicht sind. Die erzeugten Teildreiecke werden in einem Quad-Tree gespeichert, an dessen Wurzel das ursprüngliche Dreieck steht. Dreiecke mit hohen Strahlungswerten werden weiter unterteilt als Dreiecke, die z. B. im Schatten anderer Objekte liegen und somit weniger am Strahlungsaustausch teilhaben. Innerhalb dieser Phase wird Parallelität also auf Ebene der Dreiecke erzeugt.

Die zweite Phase ist die eigentliche Berechnungsphase. Dabei werden für alle Ursprungsdreiecke im BSP-Baum „Ray“-Tasks erzeugt, die den Strahlungsaustausch zwischen den Dreiecken berechnen und gegebenenfalls weitere Unterteilungen vornehmen. Diese Phase wird solange wiederholt, bis sich die Strahlungswerte der Dreiecke nicht mehr stark genug im Verhältnis zu einer vorgegebenen Fehlerschranke ändern. Innerhalb dieser Tasks wird gegebenenfalls eine große Anzahl neuer Tasks erzeugt, um weitere Unterteilungen vornehmen zu können.

Im Anschluss findet eine Nachbearbeitungsphase statt, bei der die errechneten Bildinformationen für die Grafikausgabe normalisiert werden. Dabei findet jedoch keine Interaktion mit anderen Objekten statt.

Stark vereinfacht sind die vier verschiedenen Tasktypen des hierarchischen Radiosity in

Algorithmus 3.1 dargelegt. Der „Refinement“-Task wird in der ersten Phase verwendet, um die initialen Dreiecke der Szene zu unterteilen. Dazu existiert für jedes Eingabedreieck ein solcher Task. Für jeweils ein Dreieck wird der BSP-Baum durchlaufen und Formfaktoren zwischen diesem und dem jeweiligen Dreieck des BSP-Baums berechnet. Die Formfaktoren geben den Anteil der Strahlungsmenge an, die das jeweils andere Dreieck erreicht. Er hängt sowohl von der Lage, Neigung und Größe der Objekte zu einander als auch von eventuell dazwischenliegenden anderen Dreiecken ab. Eine Fehlerschranke gibt vor, ob das größere der beiden betrachteten Dreiecke unterteilt werden muss. Da alle „Refinement“-Tasks parallel abgearbeitet werden, muss für diese Unterteilung der Eintrag im jeweiligen Quad-Tree für den Zugriff anderer Threads gesperrt werden. Da dies aber nur eines der beiden Dreiecke von einer großen Menge von Dreiecken betrifft, ist eine häufige Behinderung anderer Threads unwahrscheinlich. Die Unterteilung wird danach rekursiv für die neu erzeugten Teildreiecke wiederholt, sodass weitere Unterteilungen stattfinden können.

Schon dieser „Refinement“-Task bringt zu Beginn der Berechnungen eine Irregularität in die Berechnungen ein, da der zeitliche Ablauf Einfluss auf die durchzuführenden Operationen der Threads hat. Betrachtet beispielsweise ein Thread ein Dreieck A und ein sichtbares Dreieck B, kann der Thread aufgrund der Größenverhältnisse entscheiden, das eigene Dreieck A zu unterteilen. Hätte dagegen ein anderer Thread schon vorher die beiden interagierenden Dreiecke A und B betrachtet und entsprechend auch das Dreieck A unterteilt, würde der zuvor betrachtete Thread nicht mehr A unterteilen und stattdessen die neu erzeugten Unterdreiecke direkt untersuchen. Das heißt, ein Thread muss eventuell mehr oder weniger oft unterteilen als ein anderer Thread, wodurch eine ungleiche Lastverteilung entsteht.

Der zweite Tasktyp „Ray“-Task berechnet den eigentlichen Energieaustausch zwischen den schon unterteilten Dreiecken. Im Laufe der Unterteilung werden Interaktionen zwischen zwei Dreiecken erzeugt, falls diese untereinander zumindest teilweise sichtbar sind. Diese Interaktionen werden vom „Ray“-Task analysiert, indem die einfallende Strahlungsmenge summiert wird. Anhand einer vorgegebenen Fehlerschranke wird entschieden, ob gegebenenfalls neue Dreiecke mit entsprechend neuen Interaktionen erzeugt werden müssen.

Besitzt ein Dreieck noch nicht verarbeitete Interaktionen, werden neue „Visibility“-Task erzeugt, die jeweils für eine Menge von Interaktionen neue Sichtbarkeitsfaktoren berechnen, die im Gegensatz zu den Formfaktoren nur den gegenseitig sichtbaren Flächenanteil berechnen.

Sind alle derzeit existierenden Interaktionen bearbeitet, kann der eigentliche Strahlungsaustausch betrachtet werden. Dazu werden anhand der Interaktionenliste jedes Dreiecks die einfallenden Strahlungsanteile der jeweils anderen Dreiecke summiert. Ist das betrachtete Dreieck unterteilt, wird die Strahlungsmenge an die vier Unterdreiecke weitergegeben und für jedes Teildreieck ein eigener „Ray“-Task erzeugt. Ansonsten wird die Strahlungsmenge an das Elterndreieck gegeben. Sind alle Ergebnisse der vier Unterdreiecke vorhanden, kann das Elterndreieck die entsprechend der Gewichtung berechnete Strahlungsmenge wiederum an dessen Elterndreieck weitergeben, bis das oberste Dreieck im Quad-Tree erreicht wurde. Die Parallelität ist hier also initial durch die Menge von Basisdreiecken gegeben, aber für alle Unterdreiecke werden neue Tasks erzeugt, die in Abhängigkeit von der jeweiligen Unterteilung unterschiedlich lang rechnen. Insbesondere die Tasks der Blattknoten im Quad-Tree müssen unterschiedlich lange rechnen, da dort auch der Aufwärtslauf zur Energiebestimmung beginnt. Nur der letzte Task der jeweils vier Unterdreiecke muss das Gesamtergebnis

an das Elterndreieck übermitteln. Ist der entsprechende Task wiederum der letzte, der sein Ergebnis beisteuert, läuft er im Baum weiter nach oben als andere Tasks. Die Laufzeit einer Task ist somit nicht im Voraus bekannt und hängt auch vom zeitlichen Verlauf ab. Dies stellt somit eine Irregularität dar, die mit Lastbalancierung ausgeglichen werden muss.

Der „Visibility“-Task wird von den „Ray“-Tasks nach Bedarf erzeugt, um die Sichtbarkeitsfaktoren zweier Dreiecke zu bestimmen, also ob sie durch andere Objekte verdeckt sind und somit keinen Anteil am direkten Strahlungsaustausch haben. Wurde die Interaktionen-Liste bearbeitet, werden die weiteren Berechnungen zur Aufsammlung der Energieanteile des „Ray“-Tasks fortgeführt, d. h. der „Visibility“-Task teilt sich Berechnungen mit dem „Ray“-Task.

Der vierte Tasktyp „Average“-Task führt am Ende der Berechnungen, nachdem der Strahlungsaustausch komplett berechnet wurde, nachträgliche Korrekturen aus, um die grafische Darstellung zu verbessern. Für jedes Dreieck werden entweder neue Tasks für die jeweils vier Unterdreiecke erzeugt oder die berechneten Farbwerte gemittelt und normalisiert.

Die Implementierung des hierarchischen Radiosity ist also eine Applikation mit hoher Irregularität, die eine Lastverteilung zur Übersetzungszeit erschwert und somit spezielle Anforderungen an die dynamische Lastbalancierung stellt. Durch eine unterschiedliche Ausführungsreihenfolge der Interaktionen zwischen Objekten der Szene variiert die Gesamtzahl der verarbeiteten Interaktionen bei wiederholter Berechnung der gleichen Szene. Um diesen Störfaktor bei der Messung herauszurechnen, wird zur Leistungsbewertung nicht die absolute Zeit zur Berechnung einer Szene benutzt, sondern die Anzahl von Interaktionen pro Sekunde.

Durch die Vielzahl verschiedener Tasktypen und der großen Anzahl dynamisch erzeugter Tasks mit hoher Irregularität eignet sich diese Applikation besonders für die Untersuchungen dieser Arbeit und wird daher vertieft in späteren Abschnitten betrachtet.

Für Laufzeitexperimente werden folgende drei Szenen betrachtet:

- Szene „largeroom“:
  Anzahl von Basisdreiecken: 532
  Anzahl von Elementen: 20.976
  Anzahl von Interaktionen: 240.335

- Szene „room11“:
  Anzahl von Basisdreiecken: 2.957
  Anzahl von Elementen: 23.977
  Anzahl von Interaktionen: 2.128.270

- Szene „Halle“:
  Anzahl von Basisdreiecken: 1.157
  Anzahl von Elementen: 44.313
  Anzahl von Interaktionen: 935.507

3.5 Quicksort

Als weitere Applikation wird das „Quicksort“-Verfahren [56] als Beispiel für „Divide-and-Conquer“-Algorithmen betrachtet, dessen Parallelitätsgrad erst im Verlauf der Berechnung

steigt und somit eine spezielle Anforderung an die Lastbalancierung stellt. Zur Sortierung eines gegebenen Feldes beliebiger Elemente wird zunächst ein Pivotelement ausgewählt. In einem Partitionierungsschritt werden die Elemente schrittweise von außen nach innen so vertauscht, dass das eine Teilfeld nur Elemente kleiner oder gleich dem Pivotelement enthält, das andere Teilfeld entsprechend alle übrigen Elemente. Ist die Unterteilung abgeschlossen, wird für jedes Teilfeld der Quicksort-Algorithmus rekursiv angewendet, wodurch wiederum eine Ordnung in den Teilfeldern erzeugt wird. Nach Ende der rekursiven Abarbeitung sind alle Elemente sortiert.

In der verwendeten Implementierung wird das Testproblem durch die Vorgabe einer Feldgröße und eines Startwerts für den Zufallsgenerator definiert. Das eigentliche Feld wird dann mit Pseudo-Zufallswerten gefüllt. Die parallele Implementierung partitioniert zunächst das Feld entsprechend dem sequenziellen Algorithmus und erzeugt dann für die beiden Teilfelder jeweils einen Task für deren Sortierung. Da die Größe der beiden Teilfelder von den Eingabedaten und der jeweiligen Wahl des Pivotelements abhängt und die Teilfelder meist nicht exakt gleich groß sind, kann der Berechnungsaufwand eines Tasks nicht im Voraus angegeben werden. Selbst bei gleich großen Teilfeldern ist die Anzahl nötiger Tauschoperationen nicht vorhersagbar. Auf diese Irregularität kann mit Lastbalancierung reagiert werden. Für dieses Verfahren ist es aber auch entscheidend, die anfangs wenigen Tasks an möglichst viele Threads zu verteilen. Zu Beginn existiert nur ein einzelner Task, der das gesamte Feld partitioniert. Daher kann zu dieser Zeit kein anderer Thread Tasks bearbeiten. Erst am Ende werden zwei Tasks erzeugt, es sind also erst Tasks für genau zwei Threads verfügbar. Somit sind z. B. erst ab der fünften Unterteilung genug Tasks für 16 Threads vorhanden. Ein linearer Speedup ist daher nicht zu erreichen.

Bei einer idealen parallelen Ausführung, bei der jedes der jeweils zwei Teilfelder gleich groß ist, lässt sich der Berechnungsaufwand der parallelen Implementierung wie folgt abschätzen:

$$\begin{aligned} T_{\text{par}}(N,p) &= \sum_{i=1}^{\lceil \log p \rceil} \frac{N}{2^{i-1}} + \sum_{i=\lceil \log p \rceil + 1}^{\lceil \log N \rceil} \frac{N}{p} \\ &= 2N \left(1 - \frac{1}{2^{\lceil \log p \rceil}}\right) + \frac{N}{p}(\lceil \log N \rceil - \lceil \log p \rceil), \end{aligned} \tag{3.1}$$

wobei N die Größe des Feldes ist und p die Anzahl teilnehmender Threads. Da jeweils ein Task zwei neue Tasks erzeugen kann, sind erst ab Baumebene $\lceil \log p \rceil + 1$ genügend Tasks für alle Threads vorhanden, womit also eine vollständige parallele Abarbeitung mit allen Threads gegeben ist. Idealisiert unter der Annahme, eine gleichmäßige Verteilung wäre möglich, lässt sich also die Zeit für den Partitionierungsaufwand für die entsprechenden Baumebenen um den Faktor p reduzieren. Der erste Summand stellt den Anteil der Berechnungen dar, die nicht komplett parallel abgearbeitet werden können, da nicht genügend Tasks vorhanden sind. Mit jeder weiteren Feldunterteilung verdoppelt sich der Parallelitätsgrad, wodurch sich die nötige Zeit für die Untersuchung der N Feldelemente halbiert.

Die Abschätzung der benötigten Zeit zur parallelen Ausführung erlaubt entsprechend eine Abschätzung des unter Idealbedingungen zu erreichenden Speedups:

$$S_{\text{ideal}}(N,p) = \frac{T_{\text{seq}}(N)}{T_{\text{par}}(N,p)} = \frac{N \cdot \log N}{T_{\text{par}}(N,p)}.$$

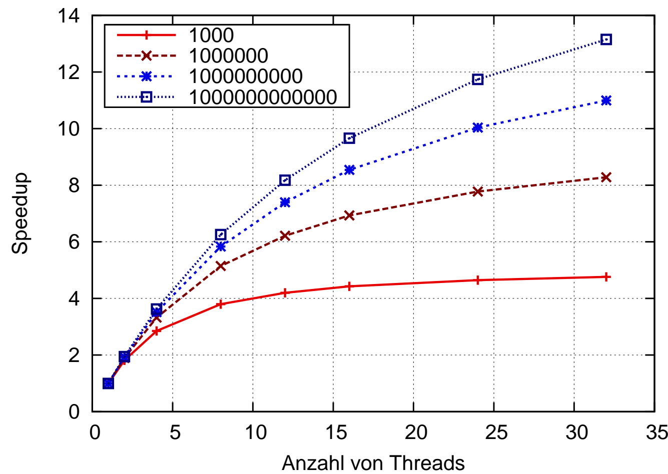


Abbildung 3.3: Idealer Speedup der parallelen Quicksort-Implementierung für verschiedene Feldgrößen.

In Abbildung 3.3 sind einige Speedup-Kurven für verschiedene Feldgrößen abgebildet. Der Speedup zur Sortierung von eintausend Feldelementen erreicht kaum 5 bei 32 Threads, aber selbst für eine Milliarde Elemente erreicht der Speedup gerade 11 für 32 Threads.

Die Werte für S_{ideal} sind jedoch nur unter Idealbedingungen zu erreichen, wenn tatsächlich in jedem Schritt eine exakte Halbierung des Feldes erreicht wird. Außerdem enthält die Abschätzung in Formel (3.1) nicht die unterschiedlichen Kosten des Partitionierungsschritts. Jedes Feldelement wird zwar einmal gelesen, ob ein Vertauschen und somit zwei Schreiboperationen notwendig sind, ist aber von der Eingabe abhängig und ist in der Formel nicht vorgesehen. Der Overhead der parallelen Taskabarbeitung ist ebenfalls nicht berücksichtigt. Der tatsächliche Speedup kann also durchaus erheblich unter dem idealen Speedup liegen. Effekte, die die parallele Ausführung verbessern, sind allerdings auch nicht berücksichtigt. So kann gerade bei der Quicksort-Applikation eine bessere Ausnutzung des Caches die Performance erhöhen, wenn Teilfelder in die Caches der einzelnen Prozessoren passen.

Um den Overhead der Taskabarbeitung zu begrenzen, werden neue Tasks nur bis zu einer bestimmten Feldgröße erzeugt. Kleinere Teilfelder werden dann innerhalb des Tasks direkt ohne Taskneuerzeugung sortiert. Im Idealfall würde es ausreichen, die Taskerzeugung auf Ebene $\log p$ zu beenden und alle weiteren Ebenen direkt innerhalb eines Tasks zu behandeln. Da aber die Feldgrößen und die damit verbundenen Berechnungen nicht fest vorhersagbar sind, ist eine weitere Unterteilung mit entsprechender Taskerzeugung sinnvoll, um damit eine Lastbalancierung zu ermöglichen. Die Wahl dieser Mindestfeldgröße hängt dabei auch von Parametern der eingesetzten Hardware ab. Synchronisationsoperationen für die Taskverwaltung benötigen wesentlich mehr Zeit als normale unsynchronisierte Speicherzugriffe. Für die späteren Laufzeit-Experimente wurde eine Mindestfeldgröße von 1000 festgelegt, die sich in Tests als genügend groß herausgestellt hat, um dennoch genügend Parallelität zu erzeugen.

Kapitel 4

Performance-Analyse Hardware-unterstützter Taskpools

In diesem Kapitel werden Hardware-Operationen für effiziente Taskpool-Implementierungen untersucht. Dazu soll festgestellt werden, wie groß das Potenzial der Hardware-nahen Implementierung von Taskpools ist. Dazu wurden alle Performance-kritischen Teile in der Hardware-nahen Assemblersprache implementiert, die eigentlichen Applikationen bleiben davon aber unberührt und nutzen weiterhin das vorgegebene Interface. Um dennoch eine gewisse Portierbarkeit zu erhalten, wurden Elemente wie Stack-Behandlung, Funktionsrahmen und ähnliche Kontrollstrukturen mittels der Makrosprache M4 [68] gekapselt.

Aufgrund der Hardware-nahen Implementierung sind die einzelnen Komponenten nicht so klar getrennt wie im vorgestelltem KOALA-Framework. Daher ist dort mit einem kleinen zusätzlichen Overhead zu rechnen. Die Aussagen über die Güte und Skalierbarkeit der jeweiligen Applikation und Taskpool-Implementierungen sind jedoch übertragbar, da die wesentliche Grundstruktur und die Ausführungshierarchie identisch ist. Zudem können prinzipiell auch einzelne Komponenten des KOALA-Frameworks Hardware-näher implementiert werden.

Darüber hinaus fließen die Ergebnisse dieser Analysen in den Entwurf der Hardware-unterstützten Komponenten ein.

Basis der Untersuchungen sind die im vorherigen Kapitel beschriebenen Applikationen, während der Schwerpunkt in der effizienten Implementierung der laufzeitkritischen Funktionen der Taskpools liegt. Dabei werden die in der Diplomarbeit des Autors [57] untersuchten Mechanismen für zusätzliche Plattformen erweitert und mit unterschiedlichen Applikationen getestet.

4.1 Einführung

Die Gesamtausführungszeit einer taskbasierten Applikation setzt sich zusammen aus dem eigentlichen Berechnungsanteil der Applikation, welcher durch die Tasks beschrieben ist und dem Anteil für die Verwaltung der Tasks innerhalb des Taskpools. Dieser Verwaltungsanteil verringert die für die Applikation zur Verfügung stehende Rechenzeit. Die Taskverwaltung stellt also einen Overhead dar, der möglichst gering sein sollte. Dieser Overhead tritt immer dann auf, wenn ein neuer Task erzeugt wurde oder ein Task beendet wurde und somit ein neuer Task zur Ausführung gewählt werden muss. Daher tritt der Overhead desto mehr in Erscheinung, desto feingranularer die auszuführenden Tasks sind. Im Wesentlichen setzt sich dabei der Overhead zusammen aus

(a) der benötigten Zeit zur Ausführung der Instruktionen zur Verwaltung des Taskpools und

(b) Wartezeiten resultierend aus der Synchronisation zwischen Threads beim Zugriff auf gemeinsame Verwaltungsdaten.

In diesem Kapitel wird die Reduzierung des Synchronisationsoverheads untersucht, indem effiziente Synchronisationsmechanismen basierend auf Hardware-Operationen [89, 91, 134] benutzt werden.

Die Synchronisation der Prozessoren beim Zugriff auf den Taskpool stellt einen wesentlichen Anteil des Overheads dar. Eine Möglichkeit, den wechselseitigen Ausschluss beim Zugriff auf gemeinsame Daten in Systemen mit gemeinsamem Speicher sicherzustellen, ist die Benutzung von Bibliotheksaufrufen oder speziellen Sprachkonstrukten. Die POSIX-Thread-Bibliothek (Pthreads) stellt entsprechende „mutex-locks“ bereit, während die Sprache Java z. B. ein „synchronized“ Schlüsselwort kennt, um spezielle Bereiche zu kennzeichnen, die jeweils nur von einem Thread betreten werden dürfen. In [69] wurden diese Methoden untersucht und miteinander verglichen.

Diese allgemeinen Methoden sind dabei mit einem Overhead verbunden, der die Abarbeitung sehr feingranularer Tasks behindert. Praktisch alle aktuellen Prozessoren unterstützen jedoch Operationen, die auf Hardware-Ebene eine schnelle Synchronisation für SMP-Systeme (Symmetric Multiprocessing) erlauben. In [6, 89] wurden entsprechend verschiedene Lock-basierte Algorithmen untersucht, um mit solchen Operationen wechselseitigen Ausschluss sicherstellen zu können. Die atomare Operation *Compare & Swap* wird z. B. in den SPARC- und Intel-Prozessoren eingesetzt. Die nicht atomare Operation *Load & Reserve* wurde in [64] vorgeschlagen und ist z. B. in MIPS- und Power-Prozessoren realisiert. Beide Klassen von Synchronisationsoperationen können zur sperrfreien Bearbeitung von Datenstrukturen genutzt werden [53, 91]. In [2] wird eine Erweiterung von *Compare & Swap* zur *DCAS*-Operation diskutiert, die eine atomare Modifikation zweier Speicherstellen erlaubt und damit das Problem der eingeschränkten Einsatzmöglichkeiten, insbesondere für nicht blockierende Modifikationen, löst. Darauf aufbauend wird in [83] die hindernisfreie (*obstruction-free*) Synchronisation paralleler Datenstrukturen diskutiert, die ein schwächeres Modell als sperrfreie oder nicht blockierende Synchronisationsoperationen realisiert. Dieser Ansatz verbessert die Performance vor allem in Fällen, in denen kein Konflikt bei Zugriffen auf gemeinsame Datenstrukturen auftritt.

Die Kosten von solchen atomaren RMW-Operationen (Read-Modify-Write) wurden z. B. in [21] auf theoretischer Grundlage analysiert. Die untersuchten Operationen werden, wie auch in dieser Arbeit, häufig zur Implementierung von Synchronisationsmechanismen eingesetzt.

Darüber hinaus werden Transaktionsspeicher [54] erforscht, um ähnlich wie Datenbanken Änderungen im Speicher durchführen und bei Bedarf verwerfen zu können. Dazu wird mit einer speziellen Operation der Beginn einer Transaktion gekennzeichnet. Alle Änderungen im Speicher sind für andere Prozessoren noch nicht sichtbar. Erst durch die Ausführung einer „commit“-Operation wird das Ende der Transaktion festgelegt, sodass die Änderungen global sichtbar werden. Falls es z. B. aufgrund konkurrierender Zugriffe anderer Prozessoren nötig ist, kann die gesamte Transaktion und somit alle Änderungen rückgängig gemacht

werden. Mit diesem Konzept können komplexere Datenstrukturen atomar modifiziert werden [7]. Transaktionsspeicher sind im Moment jedoch noch nicht kommerziell verfügbar, es existieren jedoch hardware- oder softwareorientierte Ansätze [1].

4.2 Zielarchitekturen und Synchronisationsoperationen

Für die Untersuchungen wurden folgende Systeme mit gemeinsamem Speicher betrachtet: einen Sun Fire 6800 Server mit 24 UltraSPARC-III+-Prozessoren, getaktet mit 900 MHz und einen IBM p690 Server mit 32 Power4+-Prozessoren, getaktet mit 1700 MHz. Beide Prozessorarchitekturen unterstützen jeweils eine unterschiedliche Klasse von Hardware-Synchronisationsoperationen. Neuere Prozessoren wie der UltraSPARC T1 bzw. T2 und der Power6, aber auch viele andere Prozessoren für SMP-Systeme bieten jeweils entsprechende Synchronisationsoperationen. Beide betrachteten Systeme sind SMP-Architekturen mit einer komplexen Speicherhierarchie bestehend aus einigen Cache-Ebenen und Hardware-realisierten verteilten gemeinsamem Speicher. In dem Sun-System ist der Speicher jeweils auf mehrere Prozessorenmodule verteilt, steht dem Programmierer jedoch Cache-kohärent als gemeinsamer Adressraum zur Verfügung, dessen Zugriffszeit nahezu unabhängig vom tatsächlichen Speicherort ist, also ein sogenanntes *UMA*-System (Uniform Memory Access). In dem IBM-System ist der Speicher auch über mehrere Prozessorenmodule verteilt, allerdings hängen hier die Speicherzugriffszeiten von der Distanz des zugreifenden Prozessors und des Speichermoduls ab, also ein sogenanntes *NUMA*-System (Non-Uniform Memory Access).

Um die Prozessoren eines SMP-Systems zu synchronisieren bzw. die zuverlässige Modifikation gemeinsamer Datenstrukturen zu realisieren, haben sich derzeit zwei unterschiedliche Konzepte durchgesetzt, die im Folgenden beschrieben werden.

Atomare Operationen. Sogenannte *atomare* Operationen können einen gewissen Speicherbereich in einer nicht unterbrechbaren Operation modifizieren. Soll ein Thread z. B. den Wert einer Variablen verändern, so muss der aktuelle Wert gelesen, entsprechend der Operation verändert und in den Speicher zurückgeschrieben werden (Read-Modify-Write-Zyklus; RMW). Wird der Speicherort der Variablen jedoch nach der Leseoperation von einem anderen Thread verändert, so schreibt der eigentliche Thread möglicherweise einen falschen Wert zurück. Er kann also auf konventionellem Weg nicht erkennen, ob eine Veränderung stattgefunden hat, die die Berechnungen ungültig machen. Eine atomare Operation dagegen garantiert, dass zwischen Lesen und Schreiben keine Unterbrechung stattfindet und kein anderer Thread die betreffende Speicherzelle modifizieren kann. Die auf diesem Konzept aufbauenden Systeme unterstützen üblicherweise eine Reihe unterschiedlicher atomarer Operationen.

Die einfachste Operation ist der Austausch des Wertes einer Speicherzelle (SWAP) mit einem Wert im Register des Prozessors.

Darüber hinaus erlauben Operationen nach dem „Fetch-and-Op"-Prinzip (siehe Abbildung 4.1) das Auslesen und die Modifikation einer Speicherzelle. Der gelesene Wert wird dabei innerhalb der atomaren Operation modifiziert und zurückgeschrieben. Die möglichen Operationen unterscheiden sich dabei je nach Architektur. Sehr häufig ist z. B. die atomare Addition in Form der „Fetch-and-Add"-Operation implementiert.

```
fetch_and_op( address A, register B )
{
    register T ← load(A);
    store( A, op( T, B ) );
    B ← T;
}
```

Abbildung 4.1: Pseudo-Code der atomaren *Fetch & Op*-Operation.

```
CAS( address A, register B, register C )
{
    register T ← load(A);
    if ( T == B ) store( A, C );
    C ← T;
}
```

Abbildung 4.2: Pseudo-Code der atomaren *Compare & Swap*-Operation.

Die komplexere Operation „*Compare & Swap*“ (CAS) stellt eine bedingte atomare Operation dar (Abbildung 4.2). Dabei wird ein neuer Wert aus Register C nur dann in die Speicherzelle A zurückgeschrieben, wenn im Moment des atomaren Zugriffs der vorgegebene Wert in Register B dort gespeichert ist. Folglich kann diese Operation nicht immer erfolgreich durchgeführt werden, durch entsprechende Rückgabewerte kann diese Situation aber erkannt werden und gegebenenfalls wiederholt werden, bis der Austausch erfolgreich war. Mit Hilfe dieser „*Compare & Swap*“-Operation kann also eine Speicherzelle ausgelesen werden, entsprechende Operationen können angewendet werden und deren Ergebnis wird nur dann tatsächlich atomar zurückgeschrieben werden, wenn ein definierter Wert in der Speicherzelle vorliegt. Allerdings wird eine Modifizierung einer Speicherzelle von CAS nur erkannt, wenn sich der Wert verändert hat. Steht nach einer Folge von Modifikationen in der betrachteten Speicherzelle am Ende wieder der ursprüngliche Wert, wird dies nicht erkannt. Für komplexere Datenstrukturen kann daher diese Operation oft nicht ohne weitere Modifikationen angewendet werden.

Nicht atomare Operationen. Ein anderes Konzept wird von einer Klasse von „*Load & Reserve*“- und „*Store Conditional*“-Operationen verfolgt, wie sie in [64] vorgeschlagen wurden. Die Operationen zwischen dem Laden und Speichern sind im Gegensatz zu atomaren Operationen unterbrechbar und auch Schreibzugriffe anderer Prozessoren sind erlaubt. Spezielle Schreiboperationen sind jedoch nur dann erfolgreich, wenn kein anderer Prozessor die betreffende Speicherzelle modifiziert hat. Dazu wird mit Hilfe der „*Load & Reserve*“-Operation (LR, oft auch als „*Load Linked*“ bezeichnet) eine Speicherzelle in ein Register geladen und innerhalb des Prozessors eine Reservierung auf diese Speicherzelle gespeichert. Ein Schreibzugriff eines anderen Prozessors auf diese Speicherzelle verwirft die Reservierung. Der eigentliche Prozessor kann mit Hilfe der „*Store Conditional*“-Operation (SC) einen neu berechneten Wert in diese Speicherzelle zurückschreiben. Allerdings wird diese Operation nur ausgeführt, wenn eine gültige Reservierung für die entsprechende Speicherzelle vorliegt

```
register R ← ∅; // Reservierungsregister

LR( address A, register B )
{
      R ← A; // Setze Reservierung
      B ← load(A);
}

SC( address A, register B )
{
      // Reservierung prüfen
      if ( R ≠ A ) return false;

      store(A, B);
      R ← ∅ // Reservierung löschen;
      return true;
}
```

Abbildung 4.3: Pseudo-Code der *Load & Reserve*- und *Store Conditional*-Operationen.

(siehe Abbildung 4.3). Der ausführende Prozessor kann also eine zwischenzeitliche Modifikation durch andere Prozessoren erkennen und geeignet darauf reagieren, indem z. B. die Operation wiederholt wird.

Konzeptvergleich. Die beiden Konzepte sind ähnlich leistungsfähig, da zwischen der Leseoperation und dem Zurückschreiben des neuen Wertes mit CAS oder SC beliebig komplexe Operationen realisierbar sind. Allerdings können durch das Reservierungskonzept generell Änderungen erkannt werden, wogegen CAS nur fehlschlägt, wenn ein anderer Wert als erwartet abgelegt ist. Dadurch können mittels LR/SC komplexere Datenstrukturen einfacher modifiziert werden. Zudem wird in realen Implementierungen meist die komplette Cache-Zeile reserviert, d. h. Zugriffe anderer Prozessoren auf irgendeine Speicherzelle innerhalb der gleichen Cache-Zeile der betreffenden Speicherzelle führen zu einer Invalidierung der Reservierung. Somit kann mehr als eine Speicherzelle mit dieser Klasse von Operationen überwacht werden.

Ein weiterer Nachteil der CAS-Operation ist die zusätzliche Ladeoperation und die Blockierung der Speicherzelle für die Zeit der Operation, selbst wenn die Operation fehlschlägt.

Mit den atomaren Operationen können sehr einfach und effizient Sperrvariablen realisiert werden, die im Wesentlichen nur aus einer Speicherzelle bestehen, die entweder Null oder Eins für den Status „frei“ oder „gesperrt“ enthalten. Die nicht atomaren Synchronisationsoperationen eignen sich auch für sperrfreie Datenstrukturen.

In Tabelle 4.1 sind die unterstützten Synchronisationsoperationen der in diesem Kapitel betrachteten Systeme aufgelistet. Die „UltraSPARC-III+“-Prozessoren [134] der „Sun Fire 6800“ unterstützen im Wesentlichen drei unterschiedliche atomare Operationen. Die Operation LDSTUB (*Load–Store Unsigned Byte*) erlaubt die limitierte Modifikation eines einzelnen Bytes, was für eine einfache Sperrvariable ausreicht. Die SWAP-Operation (*Swap*

Prozessor	Synchronisationsoperationen
UltraSPARC III+	SWAP, LDSTUB, CAS
Power4+	LR/SC

Tabelle 4.1: Unterstützte Synchronisationsoperationen der betrachteten Systeme.

Memory Word, auch *Fetch & Store*) tauscht den Wert einer Speicherzelle mit dem Inhalt eines Registers atomar aus. Die mächtigste Operation *Compare & Swap* (CAS) unterstützt die bedingte Modifikation einer 32-Bit- oder 64-Bit-Speicherzelle. Da diese Operationen in der SPARC-V9-Architektur vorgesehen sind, unterstützen entsprechend auch neuere Prozessoren wie der UltraSPARC-IV und T1/T2 diese Instruktionen.

Der Power4+-Prozessor [49, 124] des IBM-Systems unterstützt wie die allgemeine Power-Architektur anstatt der atomaren Operationen die nicht atomaren Operation *Load & Reserve* (LR) und *Store Conditional* (SC).

Es existieren auch Systeme, die beide Klassen von Operationen unterstützen. Die Prozessoren MIPS R10000 der SGI Origin 2000 [77] unterstützen die „LL/SC“-Operationen, aber auch die atomare Funktionsklasse *Fetch & Op*. Performance-Analysen wie in [75] zeigen, dass beiden Klassen ähnlich schnell eingesetzt werden können.

4.3 Implementierung der Taskpools

In diesem Abschnitt werden die verschiedenen Taskpool-Varianten beschrieben und deren Implementierung mit den Hardware-Operationen erläutert.

Taskpools, welche Hardware-Operationen nutzen, sind in der Assemblersprache der jeweiligen Prozessorarchitektur programmiert, um eine effiziente Benutzung der Synchronisationsoperationen zu realisieren. Da die Datenstrukturen zur Speicherung der Tasks in den betrachteten Taskpools meist aus Listen bestehen, deren Zeiger ohnehin teilweise durch die Hardware-Operationen modifiziert werden, wäre eine gemischte Implementierung in C und Assembler mit höherem Overhead verbunden.

Um den Programmieraufwand zu verringern und die Portierbarkeit zu erhöhen, wird die Makrosprache M4 benutzt, um oft verwendete Kontrollstrukturen zu kapseln.

4.3.1 Speichermanager

Der Speichermanager stellt einen wichtigen Bestandteil der Taskpool-Implementierung dar, da für jeden erzeugten Task eine Datenstruktur für dessen Argumente allokiert werden muss. Zusätzlich werden Datenstrukturen benutzt, um den Task innerhalb des Taskpools zu speichern.

Die Anzahl der Allokierungs- und Freigabeoperationen kann sehr hoch sein, während die Größe eines einzelnen Taskarguments und der entsprechenden Datenstruktur zur dessen Speicherung üblicherweise nur wenige Bytes umfassen. Dies der C-Bibliothek bzw. gegebenenfalls dem Betriebssystem zu überlassen, würde einen erheblichen Overhead bedeuten.

Da in vielen Situationen die allgemeine Speicherverwaltung des Betriebssystems eine zu lange Zeit für die Allokation benötigt (auch weil das Betriebssystem die Ressourcen unter

anderen Gesichtspunkten verwalten muss als nur eine einzelne Applikation zu berücksichtigen), wurden verschiedene skalierbare und effiziente Speichermanager entwickelt [10, 13, 78, 130, 132]. Diese können teilweise direkt als Ersatz für *malloc()* bzw. *free()* verwendet oder transparent dynamisch mit dem Programm geladen werden. Änderungen in der Applikation sind oft nicht nötig. Diese Speichermanager verwenden oft eine andere Strategie bei der Zuteilung von Speicherseiten und fragen beim Betriebssystem nur größere Blöcke an, die geeignet entsprechend der Applikationsanfragen unterteilt werden. Da diese Implementierungen jedoch einen allgemeinen Ansatz verfolgen, kann ein spezialisierter Speichermanager mit Kenntnis der Bedürfnisse des Taskpools eine höhere Performance erreichen als diese allgemeinen Konzepte. Beispielsweise werden im Taskpool meist sehr viele Datenblöcke mit identischer Größe allokiert, die eine entsprechende Optimierung z. B. mit Blockbildung erlauben.

Der Speichermanager eines Taskpools kann daher verschiedene Ansätze verfolgen, um den Overhead der Speicherbereitstellung zu reduzieren. Um die Häufigkeit von Systemaufrufen zu verringern, sollte eine Menge von Blöcken zusammen mit einem Systemaufruf allokiert werden und entsprechend stückweise an die Applikation weitergegeben werden. Ein weiterer wichtiger Punkt ist das Wiederverwenden von schon allokierten Speicherblöcken ohne Freigabe. Von der Applikation freigegebene Blöcke werden dazu in Freilisten gespeichert, aus denen direkt eine spätere Anfrage für einen neuen Speicherblock bedient wird. Diese Freilisten können zentral oder verteilt abgelegt werden, wobei bei einer zentralen Liste Synchronisation beim Zugriff darauf nötig ist. Daher ist eine Benutzung einer privaten Freiliste für jeden Thread vorteilhaft. Zwar wird so eventuell mehr Speicher allokiert als eigentlich notwendig, wenn z. B. ein Prozessor keine Elemente in seiner Freiliste mehr hat, ein anderer jedoch noch sehr viele. Allerdings bewirkt ein höherer Speicherverbrauch nicht unmittelbar eine langsamere Ausführung, die sonst nötige Synchronisation würde aber jede Operation verlangsamen.

Dieser Ansatz hat sich schon in [69] als vorteilhaft herausgestellt und wird in den folgenden Experimenten auch verfolgt. Der entsprechende Speichermanager wird mit *memdgbk* (verteilte, wachsende Liste von Blöcken) bezeichnet und wird in Kapitel 5.1.2 nochmals näher erläutert.

Die entsprechende Implementierung existiert für die Vergleichstaskpools mit Pthreads-Synchronisation in C und für die Hardware-unterstützten Taskpools in Assembler-Makrosprache.

4.3.2 Zentrale Taskpools

Zentrale Taskpools nutzen zur Speicherung der ausführungsbereiten Tasks eine zentrale Warteschlange, die von allen Threads konkurrierend zugegriffen wird. Alle Tasks stehen damit unmittelbar allen Threads zur Entnahme zur Verfügung. Allerdings kann der gemeinsame Zugriff einen Engpass darstellen, insbesondere bei feingranularen Tasks mit entsprechend häufigem Zugriff. Daher limitieren im Wesentlichen die Synchronisationsoperationen die erreichbare Skalierbarkeit.

Alle betrachteten zentralen Taskpools verarbeiten die Tasks der Warteschlange in LIFO-Ordnung (Last In, First Out), d. h. neu erzeugte Tasks werden vor älteren Tasks ausgeführt. Dies kann die Datenlokalität verbessern, da neu erzeugte Tasks eventuell auf Teile der

```
1  lock:
2          mov     1, <TReg>
3          cas     [<AddrReg>], %g0, <TReg>
4          tst     <TReg>
5          be      cont
6          nop
7  wait:
8          ld      [<AddrReg>], <TReg>
9          tst     <TReg>
10         bne     wait
11         nop
12         ba,a    lock
13 cont:
14         membar  #LoadLoad | #LoadStore
15
16 unlock:
17         membar  #StoreStore
18         membar  #LoadStore
19         st      %g0, [<AddrReg>]
```

Abbildung 4.4: Implementierung des einfachen Locks mit Null/Eins-Status für den SPARC-Prozessor.

Daten des erzeugenden Tasks zugreifen. Wird dieser Task möglichst früh ausgeführt, sind entsprechende Daten mit höherer Wahrscheinlichkeit noch im Cache zu finden.

Im Wesentlichen konzentrieren sich die Implementierungen der betrachteten zentralen Taskpools auf die benutzten Synchronisationsmechanismen. Dabei werden die folgenden Varianten untersucht: *SQ-SL* (**S**ingle **Q**ueue, **S**imple **L**ock), *SQ-TL* (**S**ingle **Q**ueue, **T**icket **L**ock) und *SQ-LF* (**S**ingle **Q**ueue, **L**ock-**F**ree). Als Vergleichsbasis wird eine Implementierung mit Pthreads-Mutex-Variablen benutzt (*SQ-PTH*).

SQ-SL (*S*ingle *Q*ueue, *S*imple *L*ock)

Diese Implementierung nutzt einen einfachen Lock-Mechanismus, der eine Variable mit Hilfe der zuvor genannten Synchronisationsoperationen auf den Wert Eins setzt, um den kritischen Bereich zu betreten. Die Operation wird fehlschlagen, wenn vorher nicht der Wert Null abgelegt war, also der Lock frei war. Die Lock-Operation wird daher den Vorgang solange wiederholen, bis die Synchronisationsoperation erfolgreich ausgeführt wurde. Der Prozessor ist also während dieser Zeit voll ausgelastet (sogenannte *busy loops*), allerdings wird in den Untersuchungen nur die Zuteilung eines Threads pro Prozessor betrachtet, sodass die Rechenzeit ohnehin nicht anderweitig in der Applikation genutzt werden könnte.

Basis des Locks ist somit eine gemeinsame Speicheradresse, die mit den Synchronisationsoperationen der entsprechenden Hardware modifiziert wird. Auf der SPARC-Architektur wird die *Compare & Swap* -Operation verwendet, um die Variable atomar zu modifizieren. Abbildung 4.4 zeigt den Aufbau der Lock-Operation. In Zeile 2-5 wird mit CAS versucht, die Speicheradresse auf den Wert Eins zu setzen, wenn zu dem Zeitpunkt eine Null dort abgespeichert ist. Anderenfalls wird in den Zeilen 8-10 gewartet, bis die Variable den Wert Null hat, der Lock also freigegeben ist. Danach wird wieder versucht, den Lock mittels CAS zu erhalten. Die Freigabe erfolgt durch Zurückschreiben einer Null in die Speicherzelle (Zeile 19). Für die Power4+-Prozessoren werden die *Load & Reserve/Store Conditional* -

```
 1  lock:
 2        lwarx    <TReg>, 0, <AddrReg>
 3        cmpwi    <TReg>, 0
 4        bne-     lock
 5        li       <TReg>, 1
 6        stwcx.   <TReg>, 0, <AddrReg>
 7        bne-     lock
 8        isync
 9
10  unlock:
11        sync
12        li       <TReg>, 0
13        stw      <TReg>, 0(<AddrReg>)
```

Abbildung 4.5: Implementierung des einfachen Locks mit Null/Eins-Status für den Power4+-Prozessor.

Operationen benutzt, um ein zu CAS ähnliches Verhalten zu realisieren (Abbildung 4.5). Dazu wird in Zeile 2-4 die LR -Operation solange wiederholt, bis der Wert Null gelesen wurde und somit der Lock als frei gilt. In Zeile 6 wird der Wert Eins zurückgeschrieben. Sollte die Operation SC fehlschlagen, weil die Reservierung nicht mehr gültig ist, wird in Zeile 7 der Versuch wiederholt. Die Freigabe erfolgt analog zur SPARC-Implementierung durch Schreiben des Wertes Null in Zeile 13.

Die zentrale Liste der ausführungsbereiten Tasks wird mit einem wie oben beschriebenen zentralen Lock geschützt. Um die Taskliste zu modifizieren, wird vorher dieser Lock des Taskpools gesperrt. Danach werden neue Tasks eingefügt oder existierende entnommen. Anschließend erfolgt die Freigabe der Lock-Variablen.

SQ-TL (*S*ingle *Q*ueue, *T*icket *L*ock)

In dieser Implementierung wird der Ticket-Lock-Mechanismus verwendet [89]. Dieser skalierbare und faire Lock basiert im Wesentlichen auf zwei Zählern. Der erste Zähler stellt das eigentliche Ticket dar. Jeder Thread, der den Lock erhalten möchte, liest diesen Wert und erhöht ihn atomar um Eins. Der zweite Zähler gibt die Nummer des Tickets an, das den kritischen Bereich betreten darf, also den Lock erhält. Der jeweilige Lock-Eigentümer gibt den Lock frei, indem er diesen zweiten Zähler erhöht, womit der Thread mit der entsprechenden nächsthöheren Ticket-Nummer den kritischen Bereich als nächstes betreten darf.

Auf SPARC-Systemen (Abbildung 4.6) wird der Ticket-Zähler mit *Compare & Swap* atomar erhöht (Zeile 2-6). Danach wird wiederholt der zweite Zähler lesend überprüft, ob das eigene, zuvor gelesene Ticket schon erreicht wurde (Zeile 9-11). Stimmt der Wert mit dem Ticket überein, erhält der Thread den Lock und kann den kritischen Bereich betreten. Die Freigabe des Locks wird durch ein einfaches Erhöhen des zweiten Zählers realisiert (Zeile 16). Da nur der Lock-Eigentümer diesen Zähler erhöht, alle anderen Thread aber nur lesend zugreifen, kann dies ohne atomare Operationen und den damit verbundenen Aufwand erfolgen.

Auf Power4+-Systemen (Abbildung 4.7) wird die Instruktionsklasse „LR/SC“ benutzt, um den mit „*Load & Reserve*“ gelesenen Wert unmittelbar nach der Erhöhung mit „*Store Conditional*“ zurückzuschreiben (Zeile 2-5).

```
1  lock:
2        ld      [<AddrRegTicket>], <TReg1>
3        add     <TReg1>, 1, <TReg2>
4        cas     [<AddrRegTicket>], <TReg1>, <TReg2>
5        cmp     <TReg1>, <TReg2>
6        bne     lock
7        nop
8  wait:
9        ld      [<AddrRegServe>], <TReg2>
10       cmp     <TReg1>, <TReg2>
11       bne     wait
12       nop
13 unlock:
14       ld      [<AddrRegServe>], <TReg1>
15       inc     <TReg1>
16       st      <TReg1>, [<AddrRegServe>]
```

Abbildung 4.6: Implementierung des Ticket-Locks für den SPARC-Prozessor.

```
1  lock:
2        lwarx   <TReg1>, 0, <AddrRegTicket>
3        addi    <TReg2>, <TReg1>, 1
4        stwcx.  <TReg2>, 0, <TReg1>
5        bne-    lock
6  wait
7        lwz     <TReg2>, 0(<AddrRegServe>)
8        cmpw    <TReg2>, <TReg1>
9        bne     wait
10       isync
11 unlock:
12       sync
13       lwz     <TReg>, 0(<AddrRegServe>)
14       addi    <TReg>, <TReg>, 1
15       stw     <TReg>, 0(<AddrRegServe>)
```

Abbildung 4.7: Implementierung des Ticket-Locks für den Power4+-Prozessor.

Im Unterschied zu dem Lock mit einfachem Null/Eins-Zustand wird beim Ticket-Lock die aufwendige Synchronisationsoperation nur solange wiederholt, bis ein gültiges Ticket ausgelesen werden konnte. Das eigentliche Warten auf die Freigabe des Locks erfolgt durch wiederholte nicht atomare Ladeoperationen. Speziell die Operation *Compare & Swap* ist aus Sicht des Speichersystems auch im fehlgeschlagenen Fall eine Speicheroperation, sodass die entsprechende Cache-Zeile in den Caches der anderen Prozessoren invalidiert wird. Der Ticket-Lock entlastet also insbesondere den Speicherbus, da wesentlich weniger Speicheroperationen ausgelöst werden. Auf Systemen mit nicht atomaren LR/SC -Operationen kann dieser Effekt weniger gravierend sein.

Der Ticket-Lock kann daher eine bessere Performance haben, allerdings gibt es auch die Möglichkeit einer verringerten Geschwindigkeit besonders auf Systemen mit hoher Last. Der Ticket-Lock als fairer Lock garantiert eine Reihenfolge bei der Zuteilung des Locks. Hat also ein Thread ein Ticket erhalten, so wird er nur auf Threads mit einer kleineren Ticket-Nummer warten müssen. Bei dem einfachen Lock *SL* kann dagegen ein „Verhungern“ auftreten, wenn sich mehrere Threads um einen Lock bemühen. Welcher Thread tatsächlich erfolgreich

die Lock-Variable auf Eins setzen kann, hängt von vielen Faktoren ab, die außerhalb der Kontrolle der Applikation liegen. Schon geringe Unterschiede bei Signallaufzeiten können einen Prozessor benachteiligen.

Der Ticket-Lock verhindert ein „Verhungern“ eines Thread bei der Lock-Anfrage, da die Lock-Zuteilungsreihenfolge durch den jeweiligen Ticket-Zähler vorgegeben ist. Ein zu einem späteren Zeitpunkt anfragender Thread erhält eine höhere Ticket-Nummer und wird somit erst nach allen bisher wartenden Threads an die Reihe kommen. Allerdings steht diesem Vorteil auch der Nachteil gegenüber, dass eine Unterbrechung eines wartenden Threads zu einer Behinderung der Threads führen wird, die eine größere Ticket-Nummer haben als der unterbrochene Thread. Dies führt zu der Situation, bei der ein Lock zwar frei ist, der anhand der Ticket-Nummer dafür zugeteilte Thread aber den kritischen Bereich aufgrund der Unterbrechung nicht betreten kann. Andere Threads können jedoch nicht stattdessen diesen Bereich betreten. In der Praxis dürfte dieses Problem aber nur dann relevant werden, wenn sich ein Thread einen Prozessor mit mehreren anderen Threads teilen muss. Für die Untersuchung wird aber eine Zuteilung eines Thread an einen Prozessor realisiert, wobei die auf der Maschine beantragten Prozessoren dediziert der Anwendung zur Verfügung stehen.

SQ-LF (*S*ingle *Q*ueue, *L*ock-*F*ree)

Diese Implementierung modifiziert die Liste ausführungsbereiter Tasks mit Hilfe eines sperrfreien Ansatzes, um den Overhead der Lock-Verarbeitung einzusparen. Die Leistungsfähigkeit eines solchen Ansatzes wurde z. B. in [91] untersucht. Die Idee ist eine atomare Modifizierung des Listenanfangs, sodass die Liste zu jeder Zeit konsistent ist.

Der sperrfreie Taskpool ist komplizierter als normale Lock-basierte Taskpools, da der Zustand der Liste zu jedem Zeitpunkt korrekt sein muss. Bei Lock-basierten Taskpools kann die Liste dagegen auch unvollständig sein, solange der zugehörige Lock gesperrt ist. Erst mit der Freigabe des Locks muss die Liste wieder konsistent sein. Beim sperrfreien Ansatz kann ein Zugriff eines anderen Threads zu einem beliebigen Zeitpunkt nicht verhindert werden.

Einfügeoperation. Das Einfügen neuer Tasks ist verhältnismäßig einfach. In die Datenstruktur des neuen Listenelements wird das aktuelle Kopfelement der Liste als Nachfolgeelement eingetragen. Dann wird das neue Listenelement atomar als neues Listenkopfelement eingetragen, wenn das zu dem Zeitpunkt eingetragene Kopfelement noch mit dem ursprünglich gelesenem Element übereinstimmt. Andererseits wird der Vorgang wiederholt.

Für den SPARC-Prozessor ist dies direkt mit der CAS -Operation erreichbar. Die nicht atomaren Operationen LR/SC des Power4+-Prozessors können nicht direkt testen, ob der Wert noch mit dem vorher gelesenen übereinstimmt. Da das Abspeichern aber fehlschlagen wird, wenn ein anderer Prozessor die Speicherzelle modifiziert hat, kann die Operation wiederholt werden, bis zwischen der Lese- und der Schreiboperation keine Änderung mehr stattfindet.

Entnahmeoperation. Das Entfernen eines Elements aus der Liste ist gegenüber dem Einfügen komplizierter. Um das neue Listenkopfelement zu bestimmen, muss das Nachfolgeelement des aktuellen Kopfelements gelesen werden. Dieses Nachfolgeelement muss dann als neues Kopfelement eingetragen werden. Um den Listenzustand konsistent zu halten,

```
 1  insert:
 2          sethi    %hi(0xffffffff), <TReg2>
 3          or       <TReg2>, %lo(0xffffffff), <TReg2>
 4  loop1:
 5          ldx      [<AddrRegList>], <TReg1>
 6          and      <TReg1>, <TReg2>, <TRegCount>
 7          srlx     <TReg1>, 32, <TRegElem>
 8          st       <TRegElem>, Offset(<AddrRegElem>,NEXT)
 9          inc      <TRegCount>
10          and      <TRegCount>, <TReg2>, <TRegCount>
11          sllx     <AddrRegElem>, 32, <TReg3>
12          or       <TRegCount>, <TReg3>, <TReg3>
13          casx     [<AddrRegList>], <TReg1>, <TReg3>
14          cmp      <TReg1>, <TReg3>
15          bne      loop1
16          nop
17
18  remove:
19          ldx      [<AddrRegList>], <TReg1>
20          and      <TReg1>, <TReg2>, <TRegCount>
21          srlx     <TReg1>, 32, <TRegElem>
22          tst      <TRegElem>
23          beq      empty
24          nop
25          ld       Offset(<TRegElem>,NEXT), <TReg4>
26          inc      <TRegCount>
27          and      <TRegCount>, <TReg2>, <TRegCount>
28          sllx     <TReg4>, 32, <TReg3>
29          or       <TRegCount>, <TReg3>, <TReg3>
30          casx     [<AddrRegList>], <TReg1>, <TReg3>
31          cmp      <TReg1>, <TReg3>
32          bne      remove
33          nop
```

Abbildung 4.8: Implementierung des sperrfreien Listenzugriffs für den SPARC-Prozessor.

darf diese Operation jedoch nur ausgeführt werden, wenn sowohl das Kopfelement als auch das Nachfolgeelement nicht verändert wurden. Die *Compare & Swap*-Operation ist dafür nicht geeignet, da nur eine Speicheradresse atomar modifiziert werden kann. Wenn beispielsweise ein anderer Prozessor das Kopfelement und dessen Nachfolgeelement entfernt und danach wieder das zuvor entfernte Kopfelement einfügt, kann die CAS-Operation des zuerst betrachteten Prozessors die Änderung des Nachfolgeelements nicht erkennen, da das Kopfelement noch übereinstimmt. Dieses sogenannte ABA-Problem [91] muss durch geeignete zusätzliche Methoden gelöst werden, um eine sperrfreie Modifikation zu realisieren.

Eine mögliche Lösung besteht in der Nutzung eines zusätzlichen Zählers, der in ungenutzten Bits der Speicheradresse abgelegt wird. In der SPARC-Architektur kann die 64-Bit-Variante der CAS-Operation benutzt werden, wenn 32-Bit-Zeiger verwendet werden. In den zweiten 32 Bit wird dann der Zähler gespeichert. Wird ein 64-Bit-Adressraum eingesetzt, kann dennoch ein Teil für einen Zähler genutzt werden, da meist nicht alle 64 Bit tatsächlich verwendet werden. Zudem können Software-seitig durch entsprechendes Alignment die niederwertigen Bits einer Adresse für diesen Zähler benutzt werden. Bei jeder Änderung der Liste muss dieser Zähler erhöht werden, sodass eine CAS-Operation eines anderen Prozessors fehlschlagen wird, selbst wenn das Kopfelement noch gleich ist. Der

```
1  insert:
2        ldarx   <TReg>, 0, <AddrRegList>
3        std     <TReg>, Offset(<AddrRegElem>,NEXT)
4        sync
5        stdcx.  <AddrRegElem>, 0, <AddrList>
6        bne-    insert
7
8  remove:
9        ldarx   <TReg1>, 0, <AddrRegList>
10       cmpdi   <TReg1>, 0
11       beq     empty
12       ld      <TReg2>, Offset(<TReg1>,NEXT)
13       stdcx.  <TReg2>, 0, <AddrRegList>
14       bne-    remove
```

Abbildung 4.9: Implementierung des sperrfreien Listenzugriffs für den Power4+-Prozessor.

Zähler wird dann unterschiedlich sein und der Prozessor muss den Vorgang wiederholen.

Dieser Ansatz löst das ABA-Problem, indem Änderungen über einen anderen Zählerstand erkannt werden können. Je geringer allerdings die zur Nutzung als Zählers zur Verfügung stehenden Anzahl von Bits ist, desto wahrscheinlicher ist es, dass Änderungen nicht erkannt werden. Durch einen Zählerüberlauf kann ein Prozessor dann wieder den zuvor gelesenen Zählerstand lesen und geht fälschlicherweise von einer unmodifizierten Liste aus. Die Nutzung freier Bits für den Zähler führt aber auch zu einem zusätzlichen Overhead, da der Zähler bei jedem Zugriff aus dem Datenwort extrahiert werden muss, entsprechend modifiziert und mit dem neuen Zeiger geeignet verknüpft und zurückgeschrieben werden muss. Dieser höhere zeitliche Aufwand kann den Vorteil einer sperrfreien Modifikation verkleinern.

Abbildung 4.8 zeigt die Implementierung dieser zählerbasierten Variante für den SPARC-Prozessor. Bei der Einfügeoperation wird in Zeile 6-8 der eigentliche Zeiger auf das erste Listenelement durch geeignete Bitmanipulation extrahiert. Die Erhöhung des Zählers und das Neukonstruieren des zusammengesetzten Datenworts erfolgt in Zeile 9-12. Die Entfernung eines Elements erfolgt durch ähnlich aufwendige Operationen, wobei die eigentliche CAS-Operation in Zeile 30 fehlschlagen wird, wenn sich der Zähler in der Zwischenzeit durch einen Schreibzugriff verändert hat. Man kann auch ohne genauere Analyse des Codes sehen, dass die eigentlich einfachere Einfügeoperation durch die Verwendung des Zählers ähnlich komplex wird wie die Entnahmeoperation.

Eine alternative Lösung des ABA-Problems kann durch Realisierung der Entnahmeoperation durch die Benutzung zweier Einfügeoperationen erreicht werden, um durch temporäres Einfügen eines NULL-Elements die komplette Liste entnehmen zu können. Dies wird in Abschnitt 4.3.3 für die sperrfreien verteilten Taskpools genauer betrachtet, da es für eine zentrale Taskliste weniger geeignet ist.

Der Power4+-Prozessor kann die nicht atomaren Operationen LR/SC besser einsetzen (Abbildung 4.9). Das Einfügen erfolgt analog der Implementierung für den SPARC-Prozessor. Das aktuelle Kopfelement wird mit der *Load & Reserve* -Operation ausgelesen (Zeile 2), entsprechend wird auf diese Speicheradresse eine Reservierung abgelegt. Das Kopfelement wird als Folgeelement des neuen Elements abgespeichert (Zeile 3). Dann wird das neue Element als neues Kopfelement eingetragen, wobei diese Operation nur erfolgreich ist, wenn

die Reservierung noch gültig ist und somit kein anderer Thread die Liste modifiziert hat (Zeile 5). Anderenfalls wird der Vorgang wiederholt. Die Speicherblöcke der Listenelemente dürfen dabei jedoch nicht in der gleichen Cache-Zeile liegen wie das Listenkopfelement, um ein Invalidieren der Reservierung zu vermeiden. In diesem Fall würde sich der Einfügevorgang unendlich oft wiederholen. Indem dem Listenkopfelement ein genügend großer Speicherblock zugeteilt wird, kann dies zuverlässig vermieden werden.

Für das Entfernen eines Elements können die Operation LR und SC erkennen, ob sich der Listenzustand verändert hat, da jede Änderung einen Schreibzugriff auf das Kopfelement bedarf. Daher kann das Entfernen ähnlich einfach realisiert werden wie das Einfügen. Mit Hilfe der *Load & Reserve* -Operation wird das aktuelle Kopfelement gelesen (Zeile 9) und eine normale Ladeoperation liest das Nachfolgeelement. Dieses Element wird dann mit SC als neues Kopfelement eingetragen (Zeile 13). Jede Änderung durch einen beliebigen Prozessor wird ein Fehlschlagen dieser Operation zur Folge haben, sodass der Vorgang wiederholt werden muss, bis ein konsistenter Zustand möglich ist.

Auch hier ist offensichtlich, dass diese Implementierung wesentlich einfacher ist als die äquivalente Implementierung auf dem SPARC-Prozessor. Gegenüber der Benutzung von Locks wurde hier tatsächlich eine Einsparung von Operationen möglich, sodass eine wesentlich bessere Performance erwartet werden kann.

SQ-PTH (*S*ingle *Q*ueue, *PTH*reads)

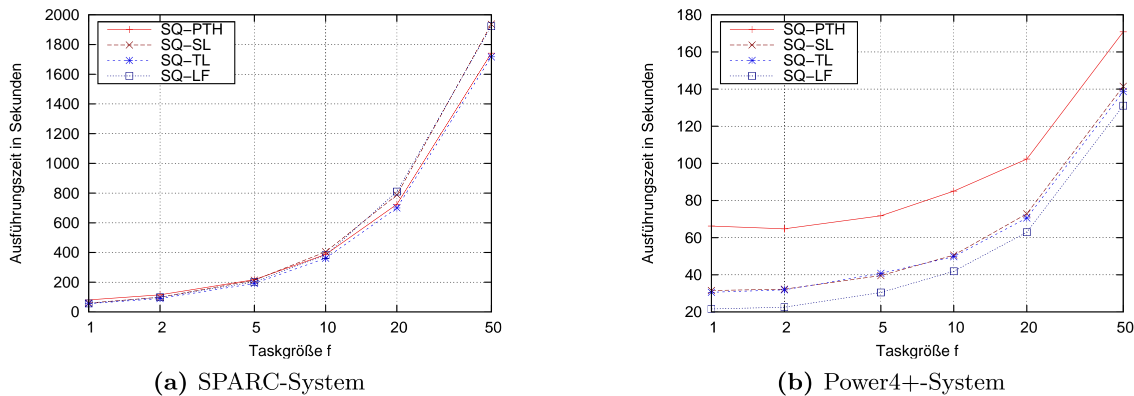
Die Vergleichsimplementierung aus [57, 69] nutzt Pthreads-Mutex-Locks, um den Zugriff auf die Liste zu kontrollieren. Somit ähnelt der Aufbau den Implementierungen *SQ-SL* oder *SQ-TL*, da vor der Modifizierung der entsprechende Lock angefragt wird. Nach Zuteilung wird die Liste modifiziert (Entnahme oder Einfügen eines Tasks) und anschließend wird der Lock wieder freigegeben.

Performance-Analyse

Für einen ersten Überblick über die Performance der zentralen Taskpools wird die synthetische Applikation, wie in Abschnitt 3.1 beschrieben, verwendet, die mittels Vorgabe der Taskgröße eine Untersuchung des Overheads erlaubt. Für den Parameter t, der die Anzahl der erzeugten Tasks steuert, wird der Wert 35 benutzt, sodass 35 initiale Tasks im Laufe der Abarbeitung über 78 Millionen Tasks erzeugen. Der Parameter f beschreibt die Taskgröße und wird als Basis der Untersuchung benutzt.

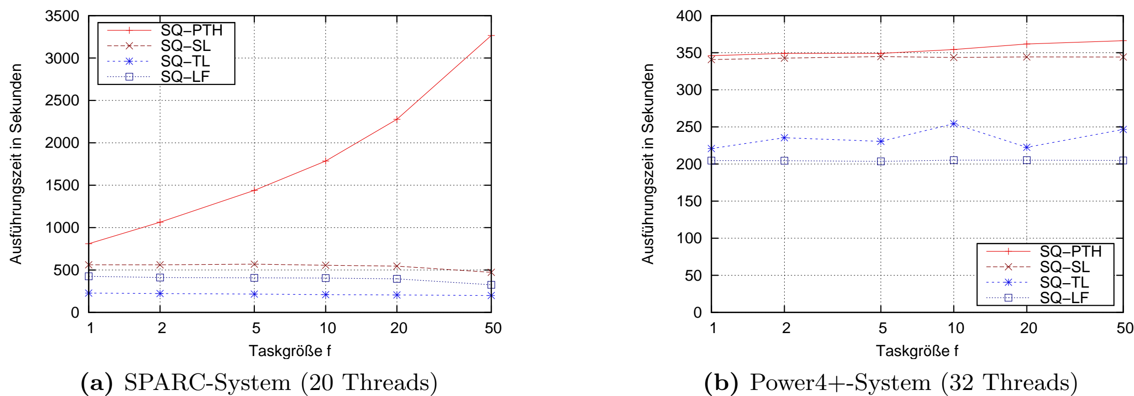
Die Ergebnisse auf den beiden betrachteten Systemen von SUN und IBM sind in den Abbildungen 4.10a bis 4.11b dargestellt. Bei der Benutzung eines Threads steigt die Ausführungszeit linear an, wenn der Faktor f, also die Taskgröße, erhöht wird. Auf der SPARC-Architektur (Abbildung 4.10a) erreichen die verschiedenen Implementierungen ähnliche Resultate. Auf dem deutlich schnelleren Power4+-System (Abbildung 4.10b) zeigt sich schon ein größerer Laufzeitunterschied. Insbesondere verlangsamt der Einsatz der Pthreads-Locks die Ausführung erheblich. Die Taskpools mit Hardware-Unterstützung sind bis zu dreimal schneller, wobei insbesondere der sperrfreie Ansatz auf diesem System schnell ist.

Bei der parallelen Abarbeitung der synthetischen Tasks zeigt sich ein anderes Verhalten. Bei 20 Threads auf dem SPARC-System (Abbildung 4.11a) kann eine erheblich langsamere



(a) SPARC-System **(b)** Power4+-System

Abbildung 4.10: Ausführungszeit der synthetischen Applikation mit zentralen Taskpools für unterschiedliche Taskgrößen (1 Thread).



(a) SPARC-System (20 Threads) **(b)** Power4+-System (32 Threads)

Abbildung 4.11: Ausführungszeit der synthetischen Applikation mit zentralen Taskpools für unterschiedliche Taskgrößen (parallele Ausführung).

Ausführung der Pthreads-Implementierung gegenüber den Implementierungen mit Hardware-Operationen gemessen werden. Allerdings skaliert selbst die schnellste Implementierung *SQ-TL* bei Einsatz des Ticket-Locks überhaupt nicht und erreicht z. B. bei einer Taskgröße von $f = 5$ gerade die Laufzeit der sequenziellen Variante, anstatt die Laufzeit um den Faktor 20 zu verkleinern. Der Overhead der Pthreads-Locks ist so hoch, dass sich die Laufzeit erst mit zehn mal größeren Tasks als im Diagramm gezeigt ($f \approx 500$) der Laufzeit der Hardware-unterstützten Taskpools angleicht.

Auf dem Power4+-System zeigt sich das gleiche Skalierungsproblem (Abbildung 4.11b). Die schlechtesten Ergebnisse erreicht die Pthreads-Implementierung. Aber selbst die beste Implementierung mit sperrfreien Listen ist wesentlich langsamer als die sequenzielle Variante. Dennoch ist ersichtlich, dass auf dieser Plattform der sperrfreie Ansatz prinzipiell gut funktioniert, da er effizient mit den nicht atomaren Synchronisationsoperationen umgesetzt werden kann.

Tabelle 4.2 zeigt eine Übersicht der höchsten Speedups der einzelnen Implementierungen mit einer festen Taskgröße $f = 50$. Basis der Speedup-Berechnung ist eine sequenzielle

Taskpool-Implementierung	SPARC			Power4+		
	Höchster Speedup	Anzahl Threads	Effizienz	Höchster Speedup	Anzahl Threads	Effizienz
SQ-PTH	3,1	4	**0,78**	0,95	2	0,47
SQ-SL	9,1	12	0,76	1,06	2	0,53
SQ-TL	8,98	16	0,56	1,13	2	**0,57**
SQ-LF	**11,26**	16	0,70	**1,7**	4	0,43

Tabelle 4.2: Speedup und Effizienz der zentralen Taskpools mit einer Taskgröße $f = 50$ auf dem SPARC- und dem Power4+-System.

Variante des Taskpools, die keinerlei Synchronisationsoperationen durchführt und somit den geringsten Overhead aufweist, aber natürlich parallel nicht gut funktionieren kann. Auf dem SPARC-System wurde der höchste Speedup von der sperrfreien Implementierung erreicht, allerdings mit einer Beschränkung auf 16 von 20 möglichen Threads. Dennoch ist die Effizienz mit $0,7$ nicht besonders hoch.

Das schnellere Power4+-System hat einen größeren Abstand zwischen CPU-Geschwindigkeit und Speicherzugriffszeit. Daher sind die gleichen Tasks schneller abgearbeitet und das Verhältnis der Berechnungen gegenüber der Speicherzugriffszeit bei der Verwaltung der Tasks sinkt. Entsprechend schlecht funktionieren die zentralen Taskpools. Der beste Speedup ist gerade einmal $1,7$ bei nur 4 genutzten Threads von maximal 32 möglichen. Zentrale Tasklisten sind also nur bei wesentlich größeren Tasks einsetzbar.

Als Extremwert gilt die Betrachtung leerer Tasks, die also keinerlei Berechnungen ausführen ($f = 0$). Somit wird nur der Verwaltungsaufwand gemessen. In diesem Szenario kann keine Implementierung einen Speedup über 1 erreichen, unabhängig von der Anzahl verwendeter Threads. Der höchste Speedup auf dem SPARC-System ist dort $0,52$ und auf dem Power4+-System $0,33$, jeweils für die sperrfreien Implementierungen.

4.3.3 Verteilte Taskpools ohne Taskgruppierung

Zentrale Taskpools sind sehr einfach, stellen aber durch die zentrale Listenverwaltung einen Flaschenhals dar, wenn durch viele kleine Tasks besonders häufig auf den Taskpool zugegriffen wird. Die Benutzung von mehreren Warteschlangen verteilt auf mehrere Threads verringert die Wahrscheinlichkeit von Konfliktzugriffen auf eine Liste, da weniger Threads gleichzeitig darauf zu greifen. Da die Tasks allerdings auf mehrere Listen verteilt sind, müssen zur Entnahme gegebenenfalls mehrere Listen untersucht werden. Lastbalancierungsverfahren versuchen, die ausführungsbereiten Tasks derart auf die Listen zu verteilen, dass Threads möglichst wenig auf Listen anderer Threads mit entsprechend höherer Konfliktwahrscheinlichkeit zugreifen müssen. Entnimmt ein Thread Tasks anderer Listen, wenn seine eigene Taskliste leer ist, so nennt man den Vorgang „Task-Stealing“. Diese Zugriffe sollten jedoch durch geeignete Lastbalancierungsverfahren vermieden oder reduziert werden.

Die Verwendung verteilter Tasklisten kann darüber hinaus auch die Datenlokalität verbessern, da neu erzeugte Tasks in der Liste des erzeugenden Threads gespeichert werden. Diese werden somit auch mit hoher Wahrscheinlichkeit von dem erzeugenden Thread ausgeführt,

sodass verwendete Daten aus vorherigen Tasks eventuell noch im Cache des zugehörigen Prozessors sind. Dagegen ist es weniger wahrscheinlich, dass Daten für gestohlene Tasks im Cache des stehlenden Threads sind.

Prinzipiell kann eine beliebige Anzahl von Tasklisten verwendet werden, die den Threads zugeordnet werden. Die Synchronisationsmechanismen funktionieren jedoch am besten, wenn es keine Kollision mehrerer Threads gibt. Deshalb wird bei den hier vorgestellten Implementierungen jedem Thread genau eine Datenstruktur zugewiesen, sodass ein Thread neue Tasks nur in seine eigene Liste einfügt und auch primär aus dieser Liste ausführungsbereite Tasks entnimmt. Sind durch geeignete Lastbalancierungsverfahren nach Möglichkeit die Listen zu jeden Zeitpunkt gefüllt, muss kein Thread auf Listen anderer Threads zugreifen und es treten keine Kollisionen bei der Lock-Anfrage auf.

Muss dennoch nach Tasks in fremden Listen gesucht werden, so geht jeder Thread mit einer leeren Taskliste anhand eines Feldes mit permutierten IDs der anderen Threads in jeweils unterschiedlicher Reihenfolge die einzelnen Listen der anderen Threads durch.

Die hier betrachteten Taskpool-Implementierungen speichern einzelne Tasks als kleinste Einheit ab, wie dies auch in den zentralen Taskpools gemacht wird. Eine Gruppierung mehrerer Tasks kann den Synchronisationsaufwand verringern, führt aber zu anderen Einschränkungen. Dies wird im folgenden Abschnitt 4.3.4 betrachtet.

Im Folgenden werden die verschiedenen untersuchten Implementierungen und die dafür eingesetzten Synchronisationsverfahren beschrieben.

DQ-SL (*D*istributed *Q*ueue, *S*imple *L*ock)

Diese Implementierung schützt die jeweiligen Listen mit Hilfe des einfachen Locks, der einen Null/Eins-Zustand mit *Compare & Swap* oder *Load & Reserve/Store Conditional* realisiert. Neue Tasks werden in die dem Thread zugeordnete Liste eingetragen. Bei der Taskentnahme wird diese Liste auch zuerst berücksichtigt. Ist sie leer, werden die Listen der anderen Threads entsprechend der Reihenfolge im Permutationsfeld nach Tasks durchsucht. Sowohl beim Einfügen als auch beim Entnehmen eines Tasks muss der Lock für die entsprechende Liste angefragt werden.

DQ-SL-PRIV (*D*istributed *Q*ueue, *S*imple *L*ock, *PRIV*ate buffer)

Zusätzlich zur zuvor genannten Implementierung *DQ-SL* wird hier eine private Taskliste für jeden Thread abgespeichert, die nur für den zugeordneten Thread zugänglich ist. Neue Tasks werden in dieser Liste abgelegt und werden erst in die öffentliche Liste überführt, wenn die private Liste eine gewisse Länge überschritten hat. Entsprechend werden zuerst Tasks aus der privaten Liste entnommen, bevor die öffentliche Liste durchsucht wird. Der Zugriff auf die private Liste erfordert keinerlei Synchronisation und kann so die Geschwindigkeit verbessern. Allerdings stehen die privaten Tasks nicht für andere unbeschäftigte Threads zum Stehlen zur Verfügung und verringern so die verfügbare Parallelität. Um diesen Effekt zu beschränken, wird in den Untersuchungen für alle mit **PRIV** markierten Implementierungen ein einzelner privater Listeneintrag realisiert.

DQ-SL-PRIV-C (*D*istributed *Q*ueue, *S*imple *L*ock, *PRIV*ate buffer, *C*ache optimized)

Die Cache-optimierten Varianten (markiert mit **C**) richten die einzelnen Datenstrukturelemente derart im Speicher aus, dass die Zeiger für eventuelle private Listen, öffentliche Listen, Lock-Variablen und weitere Daten auf verschiedene Cache-Zeilen fallen. Insbesondere die Hardware-unterstützten Synchronisationsoperationen haben oft Auswirkungen auf eine ganze Cache-Zeile, sodass eine entsprechend optimierte Abspeicherung Cache-Invalidierungen durch Schreiboperationen verhindert, die nicht zu der eigentlichen Synchronisation gehören. Auf dem Power4+-System mit dem Reservierungskonzept kann dies erheblichen Einfluss auf die Performance der Synchronisationsoperationen haben. Allerdings sind bei diesem Ansatz zusätzliche Speichertransfers notwendig, da nicht mehr alle Daten in einer Cache-Zeile liegen und somit nicht mit einem einzigen Zugriff in den Cache geladen werden.

DQ-SL-TRY-PRIV-C (*D*istributed *Q*ueue, *S*imple *L*ock, only single *TRY*, *PRIV*. buffer, *C*ache opt.)

Diese Variante modifiziert das Verhalten beim Stehlen fremder Tasks, wenn die eigene Taskliste leer ist. Dabei wird ein semi nicht blockierender Ansatz verfolgt. Der Thread versucht dabei nur einmal, den Lock eines anderen Threads zu erhalten. Schlägt also die *Compare & Swap*- oder *Store Conditional*-Operation fehl, wird unmittelbar die Liste des nächsten Threads untersucht. Dies kann insbesondere dann zu einer verbesserten Performance führen, wenn mehrere Threads häufig nach neuen Tasks suchen und durch die Suchreihenfolge die gleiche Taskliste eines Threads zugreifen. Bei diesem Ansatz erhält nur einer der Threads den Zugriff, eventuell andere zugreifende Threads suchen stattdessen bei den jeweiligen Folgethreads nach Tasks.

DQ-TL-PRIV-C (*D*istributed *Q*ueue, *T*icket *L*ock, *PRIV*. buffer, *C*ache opt.)

Diese Implementierung setzt den zuvor schon erläuterten Ticket-Lock ein, um den Zugriff auf die jeweiligen Tasklisten zu kontrollieren.

DQ-CLH-PRIV-C (*D*istributed *Q*ueue, *CLH* lock, *PRIV*. buffer, *C*ache opt.)

Ein interessanter Vertreter der fairen Locks ist der CLH-Lock [88]. Diese Implementierung reduziert die aufgrund der „busy-loops“ benötigte Speicherbandbreite und stellt darüber hinaus geringere Anforderungen an den Prozessor. Der CLH-Lock realisiert eine implizite Liste der wartenden Threads. Jeder Thread besitzt dabei einen Speicherblock, der u. a. einen Zeiger auf den vorherigen Thread in der Warteliste enthält. Als Lock dient ein dediziertes Speicherelement, das einen Zeiger auf das Kopfelement enthält, welches anfangs auf ein Dummy-Element zeigt.

Um den Lock zu erhalten, setzt der Thread zunächst in seinem Speicherblock eine Wartevariable auf einen spezifischen Wert, um damit nachfolgende Threads zum Warten zu bringen. Dieser Speicherblock wird dann atomar mit dem Element vertauscht, auf das der Zeiger im Lock-Speicherblock zeigt. Der Thread wartet auf die Freigabe des Locks, indem die Wartevariable des vertauschten Speicherblocks wiederholt gelesen wird. Die Freigabe

des Locks erfolgt durch Löschen der Wartevariable des vor dem Vertauschen gespeicherten Speicherblocks. Ein nochmaliges Vertauschen bei der Freigabe erfolgt jedoch nicht, der getauschte Speicherblock wird für spätere Lock-Anfragen gespeichert. Sobald der Eigentümer des ausgetauschten Speicherblocks den Lock freigibt, kann der Thread den kritischen Bereich betreten.

Bei n Threads existieren somit $n+1$ Speicherblöcke, wobei ein Thread genau einen Block besitzt und ein weiterer Block global verfügbar ist. Bei einer Lock-Operation wird der eigene Speicherblock mit dem globalen Block vertauscht. Der Thread erhält den vorherigen Block aber bei Freigabe nicht zurück sondern benutzt den erhaltenen Block für weitere Operationen. Dadurch besitzt kein Thread einen Block exklusiv.

Der CLH-Lock reduziert die Cache-Zeileninvalidierungen durch Schreibzugriffe auf ein Minimum, da nur eine SWAP-Operation zu Beginn nötig ist, die bei atomarer Implementierung nicht fehlschlägt. Darüber hinaus wird nur bei der Freigabe des Locks eine Cache-Zeile genau eines Threads beschrieben, alle anderen wartenden Threads bemerken davon nichts.

Auf dem SPARC-System ist ein „Verhungern“ nicht mehr möglich, da die SWAP-Operation gegenüber der *Compare & Swap*-Operation nicht fehlschlägt und so nach genau einer atomaren Operation die Wartephase betreten werden kann. Auf dem Power4+-System muss dagegen das Funktionspaar LR/SC sooft wiederholt werden, bis das Vertauschen erfolgreich war.

Trotz eines relativ komplizierten Aufbaus ist der tatsächliche Programmcode nicht umfangreicher als der einfache Lock oder der Ticket-Lock.

DQ-SL-SP-PRIV-C (***D***istributed ***Q***ueue, ***S***imple ***L***ock with ***S***ingle***P***ut extension, ***PRIV***. buffer, ***C***ache opt.)

Diese Implementierung nutzt aus, dass bei den verteilten Taskpools nur der Tasklisten-Eigentümer neue Tasks einfügt. Die Idee ist dabei, dass diese Einfügeoperationen komplett ohne Synchronisationsoperationen auskommen und nur bei der Entnahme entsprechend wechselseitiger Ausschluss sichergestellt werden muss.

Um dies zu realisieren, wird die Ordnung der Liste von LIFO (*L*ast *I*n, *F*irst *O*ut) auf FIFO (*F*irst *I*n, *F*irst *O*ut) umgestellt. Neue Tasks werden am Ende der Liste eingefügt, während vorhandene Tasks am Anfang der Liste entnommen werden. Somit werden ältere Tasks zuerst ausgeführt. Durch Verwendung eines Dummy-Elements, sodass die Liste nie leer werden kann, kann der Eigentümer der Liste neue Tasks ohne Synchronisation einfügen. Das letzte Element stellt dabei jeweils das Dummy-Element dar, das nicht entnommen werden darf. Der Thread speichert also beim Einfügen die Daten des neuen Tasks in dieser Struktur ab und fügt ein neues Dummy-Element an. Daraufhin kann die Entnahme des vorbereiteten Tasks auch durch andere Threads erfolgen.

Zur Entnahme müssen jedoch sowohl der Listenbesitzer als auch stehlende Threads wie zuvor einen Lock für den Anfang der Liste anfragen. In dieser Implementierung wird der *SimpleLock* (SL) verwendet.

Die Vorteile beim Einfügen von Tasks können allerdings durch verschlechterte Datenlokalität beim Ausführen von Tasks zunichte gemacht werden. Da ältere Tasks zuerst ausgeführt werden, steigt die Wahrscheinlichkeit, dass dafür nötige Daten nicht mehr im Cache vor-

handen sind, wodurch mehr Cache-Nachladeoperationen auftreten können als bei LIFO-Ausführung.

DQ-LF-PRIV-C (*D*istributed *Q*ueue, *L*ock-*F*ree, *PRIV*. buffer, *C*ache opt.)

Auch für die verteilten Taskpools wird hier ein sperrfreier Ansatz verfolgt. Dabei wird im Wesentlichen der gleiche Ansatz wie bei der zentralen Taskliste in Abschnitt 4.3.2 verwendet. Für den Power4+-Prozessor werden die Listen mit nicht atomaren Operationen sperrfrei modifiziert, während für den SPARC-Prozessor der zählerbasierte Ansatz verfolgt wird.

DQ-LF-SPARC2-PRIV-C (*D*istributed *Q*ueue, *2*.lock-free for *SPARC*, *PRIV*. buffer, *C*ache opt.)

Aufgrund der genannten Einschränkungen bei der sperrfreien Modifikation auf der SPARC-Architektur unter Benutzung der atomaren Synchronisationsoperationen wird hier ein weiterer Ansatz untersucht.

Die Benutzung eines Zählers zur Erkennung der Listenmodifikation beim Entfernen eines Tasks erhöht den Aufwand auch bei kollisionsfreien Zugriffen erheblich. Durch die Verwendung verteilter Tasklisten kann aber von einer reduzierten Wahrscheinlichkeit einer Kollision ausgegangen werden, da im Idealfall kein Task-Stealing notwendig ist.

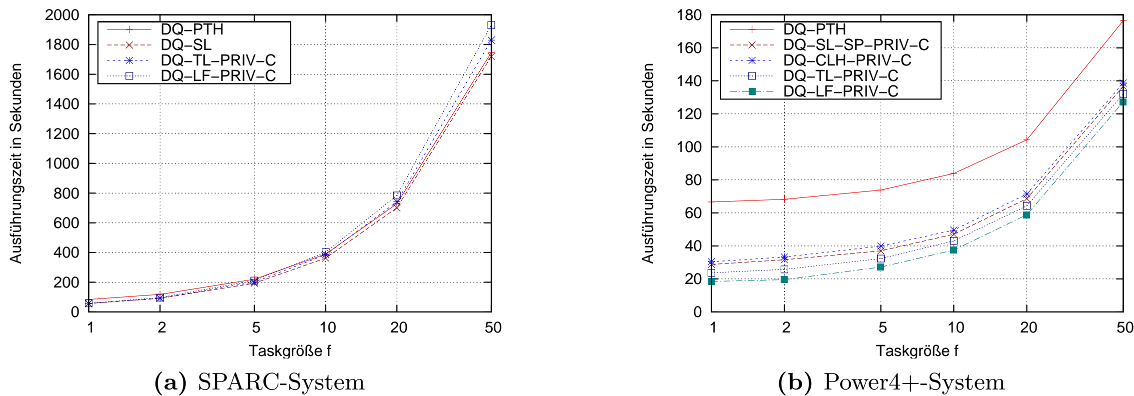
In der alternativen Implementierung **LF-SPARC2** wird die Entnahmeoperation durch zwei Einfügeoperationen ersetzt. Im ersten Schritt wird die gesamte Liste entfernt, indem das Listenkopfelement auf *NULL* gesetzt wird. Wie bei der Einfügeoperation wird also das aktuelle Listen-Kopfelement ausgelesen, allerdings nur lokal gespeichert und durch *NULL* ersetzt. Dies kann sehr einfach mit *Compare & Swap* erreicht werden. Nun kann der Thread ohne weitere Synchronisationsmechanismen ein Listenelement entfernen.

Anschließend wird die so modifizierte Liste wieder als neues Kopfelement eingetragen, wenn das aktuelle Element immer noch *NULL* ist. Dies kann auch unmittelbar mit *Compare & Swap* erfolgen. Ist die öffentliche Liste jedoch nicht mehr leer, weil z. B. der Listeneigentümer neue Tasks eingefügt hat, muss die neue Liste zuvor erneut wie oben beschrieben entfernt werden und mit der noch lokalen Liste vereint werden. Der Einfügevorgang wird auf diese Art solange wiederholt, bis das Vertauschen erfolgreich war.

Da das Neueinfügen der kritische Punkt ist, bei dem möglichst kein Listenverschmelzen nötig sein sollte, ist dieses Verfahren bei einer zentralen Liste nicht sinnvoll. Dort ist die Wahrscheinlichkeit sehr hoch, dass irgendein anderer Thread neue Tasks eingefügt hat. Dies könnte dazu führen, dass die lokale Zwischenliste immer mehr Tasks enthält, während andere Threads neue Tasks suchen. Bei verteilten Tasklisten kann nur der Eigentümer neue Tasks einfügen. Somit tritt das Problem des wiederholten Neueinfügens nur auf, wenn andere Threads Tasks stehlen wollen. Wird dies durch geeignete Lastbalancierungsverfahren reduziert, kann diese Implementierung gute Ergebnisse liefern.

Dieser Ansatz hat zudem den Vorteil, keine zusätzlichen Bits zu benötigen und ohne möglichen Überlauf immer korrekt und universell zu funktionieren.

Da dieser Ansatz auf der Power4+-Architektur aufgrund der flexibleren nicht atomaren Synchronisationsoperationen nicht nötig ist, wurde dieser Ansatz nur auf dem SPARC-System implementiert.



(a) SPARC-System

(b) Power4+-System

Abbildung 4.12: Ausführungszeit der synthetischen Applikation mit verteilten Taskpools für unterschiedliche Taskgrößen (1 Thread).

DQ-PTH (*D*istributed *Q*ueue, *PTH*reads)

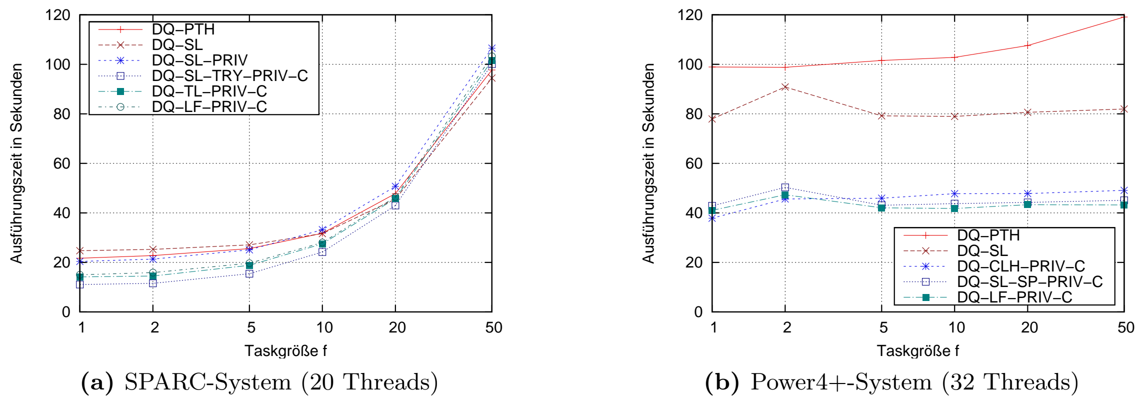
Diese Vergleichsimplementierung verwendet Pthreads-Mutex-Locks, um den Zugriff auf die verteilten Listen zu kontrollieren. Vor der Modifizierung wird der zur entsprechenden Liste gehörende Lock angefragt. Nach Zuteilung können Tasks entnommen oder eingefügt werden und anschließend wird der Lock wieder freigegeben.

Performance-Analyse

Die Laufzeitexperimente mit der synthetischen Applikation zeigen, dass die verteilten Taskpools generell eine bessere Performance als zentrale Taskpools erreichen. Abbildung 4.12a zeigt die Laufzeiten der verteilten Taskpools unter Benutzung eines einzelnen Threads auf dem SPARC-System. Die Laufzeiten der Implementierungen sind ähnlich, allerdings sind die optimierten Varianten bei kleinen Tasks ($f < 10$) bis zu einem Faktor von $2,6$ schneller. Auf dem Power4+-System in Abbildung 4.12b zeigt sich wie zuvor bei den zentralen Taskpools, dass der Abstand zu der Implementierung mit Pthreads-Locks wesentlich größer ist. Die Pthreads-Implementierung ist im schlechtesten Fall fast $4,5$-mal langsamer als die sperrfreie Implementierung mit Hardware-Unterstützung. Auch bei größeren Tasks ist die Pthreads-Implementierung deutlich langsamer. Erst mit wesentlich größeren Tasks gleicht sich die Laufzeit der Implementierung den Hardware-optimierten Implementierungen an.

Bei der parallelen Taskabarbeitung erreichen die verteilten Taskpools eine wesentlich bessere Laufzeit. Die Messergebnisse auf dem SPARC-System in Abbildung 4.13a zeigen einen linearen Verlauf in Abhängigkeit von der Taskgröße f, außer für sehr feingranulare Tasks mit $f < 10$. Dabei ist die beste Implementierung mit dem semi nicht blockierenden Taskpool um den Faktor $1,91$ schneller als die Pthreads-Implementierung. Auch die Implementierung mit dem Ticket-Lock erzielt geringe Laufzeiten und ist schneller als beide sperrfreien Implementierungen. Der für die sperrfreie Modifikation der Tasklisten nötige zusätzliche Aufwand ist größer als die Einsparung durch den Verzicht auf explizite Lock-Operationen.

Die Messungen auf dem Power4+-System (Abbildung 4.13b) zeigen, dass der sperrfreie



(a) SPARC-System (20 Threads) **(b)** Power4+-System (32 Threads)

Abbildung 4.13: Ausführungszeit der synthetischen Applikation mit verteilten Taskpools für unterschiedliche Taskgrößen (parallele Ausführung).

Ansatz hier die besten Laufzeiten erreicht und wesentlich schneller als die Pthreads-Implementierung ist. Allerdings ist der Verwaltungsoverhead noch so groß, dass alle Implementierungen keinen linearen Speedup erreichen und für sehr kleine Tasks sogar langsamer als in der sequenziellen Ausführung sind.

Der CLH-Lock ist zwar schneller als der einfache Lock, allerdings erreicht der ebenfalls faire Ticket-Lock eine höhere Performance. Auf diesem System lassen sich die nicht atomaren Operationen *Load & Reserve/Store Conditional* jedoch besser einsetzen, was sich in der guten Performance der sperrfreien Taskpools zeigt.

In Tabelle 4.3 sind die besten Speedups der verschiedenen Implementierungen beider Systeme für eine gewählte Taskgröße von $f = 50$ zusammengefasst. Für diese etwas größeren Tasks werden auf dem SPARC-System Speedups von bis zu $18,09$ erreicht, was bei der Benutzung von 20 Threads einer Effizienz von $0,9$ entspricht. Auf dem Power4+-System dagegen erreicht die beste Implementierung mit Hilfe des sperrfreien Ansatzes gerade einmal einen Speedup von $3,25$, wobei der Einsatz von mehr als 4 Threads keine weitere Verbesserung brachte. Der Overhead muss also weiter reduziert werden.

4.3.4 Blockverteilte Taskpools

Die bisherigen Experimente zeigten, dass Tasks gleicher Größe auf dem schnelleren IBM-System nicht so gut handhabbar sind wie auf dem SPARC-System. Mit steigender Rechenleistung wird der Abstand zwischen CPU-Geschwindigkeit und Speichergeschwindigkeit größer, daher sollten weitere Optimierungen vorgenommen werden, um Speicherzugriffe auf den Hauptspeicher zu verringern und so die Effizienz der Taskpools zu erhöhen. Um den Overhead bei häufigem Zugriff auf die Taskpools und bei häufigem Task-Stealing zu reduzieren, werden in den blockverteilten Taskpools die Tasks zu Gruppen zusammengefasst, die jeweils als Einheit bei Bedarf zu anderen Threads verschoben werden können. Damit können mit einer Operation direkt mehrere Tasks entnommen werden. Außerdem kann dies einen positiven Effekt auf die Datenlokalität haben, da die Tasks eines Blocks mit höherer Wahrscheinlichkeit auf gemeinsame Daten zugreifen als einzeln entnommene Tasks. Wenn aber nur ganze Blöcke von Tasks gestohlen werden können, kann sich die

Taskpool-Implementierung	SPARC			Power4+		
	Höchster Speedup	Anzahl Threads	Effizienz	Höchster Speedup	Anzahl Threads	Effizienz
DQ-PTH	17,49	20	0,87	1,32	4	0,33
DQ-SL	**18,09**	20	**0,90**	1,47	32	0,05
DQ-SL-PRIV	16,04	20	0,80	2,68	32	0,08
DQ-SL-PRIV-C	17,22	20	0,86	3,07	8	0,38
DQ-SL-TRY-PRIV-C	17,07	20	0,85	3,1	4	0,78
DQ-TL-PRIV-C	16,83	20	0,84	2,91	4	0,73
DQ-CLH-PRIV-C	17,04	20	0,85	2,88	4	0,72
DQ-SL-SP-PRIV-C	16,86	20	0,84	2,91	8	0,36
DQ-LF-PRIV-C	16,55	20	0,83	**3,25**	4	**0,81**
DQ-LF-SPARC2-PRIV-C	15,86	20	0,79	N/A	N/A	N/A

Tabelle 4.3: Speedup und Effizienz der verteilten Taskpools mit einer Taskgröße $f = 50$ auf dem SPARC- und dem Power4+-System.

Lastbalance verschlechtern, da die Tasks eines Blocks nicht mehr für andere Threads zur Verfügung stehen.

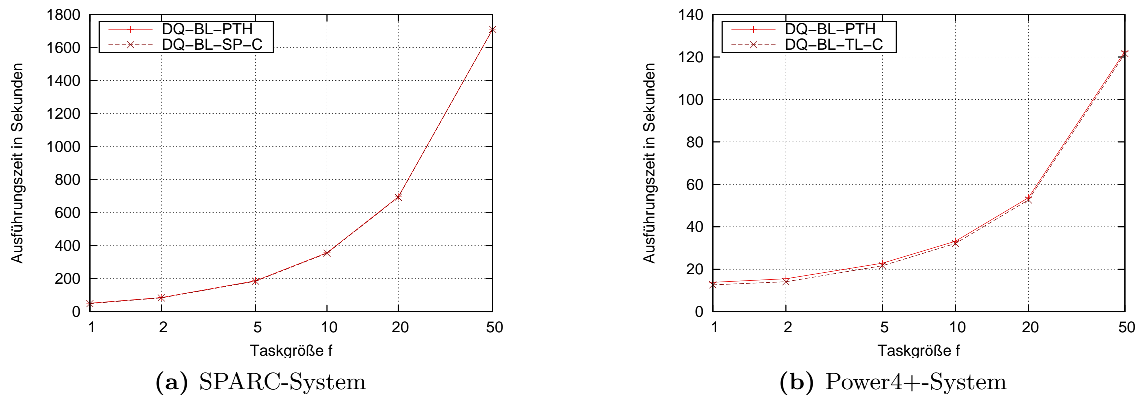
In den betrachteten Implementierungen wird daher eine kleine Gruppengröße von vier Tasks pro Block gewählt. Dabei werden zwei Blöcke in einer privaten Liste gespeichert, die vom der Liste zugeordneten Thread ohne Synchronisationsoperationen modifiziert werden kann. Der Thread trägt neue Tasks in einem Block im privaten Bereich ein. Sobald beide Blöcke gefüllt sind, wird ein Block davon in die öffentliche Liste eingetragen, die von einem entsprechenden Synchronisationsmechanismus geschützt ist. Im Falle des Task-Stealings wird ein Block von einem anderen Thread entnommen und im eigenen privaten Bereich gespeichert. Darüber hinaus arbeiten die Taskpool-Implementierungen wie die verteilten Taskpools ohne Gruppierung. Folgende Implementierungen wurden dabei betrachtet:

DQ-BL-SL-C (*BL*ock-*D*istributed *Q*ueue, *S*imple *L*ock, *C*ache opt.)

Analog zu *DQ-SL-PRIV-C* setzt diese Implementierung den einfachen Lock auf Null/Eins-Basis ein, um den Zugriff auf die öffentliche Liste des jeweiligen Threads zu schützen. Im privaten Bereich werden bis zu zwei Blöcke abgespeichert.

DQ-BL-SL-TRY-C (*BL*ock-*D*istributed *Q*ueue, *S*imple *L*ock, only single *TRY*, *C*ache opt.)

Wie in der vorherigen Implementierung *DQ-BL-SL-C* werden hier bis zu zwei Blöcke im privaten Bereich gespeichert. Zusätzlich wird der semi nicht blockierende Ansatz verfolgt, bei dem beim Stehlen nur einmal versucht wird, den Lock der entsprechenden Taskliste zu erhalten.



(a) SPARC-System

(b) Power4+-System

Abbildung 4.14: Ausführungszeit der synthetischen Applikation mit blockverteilten Taskpools für unterschiedliche Taskgrößen (1 Thread).

DQ-BL-TL-C (*BL*ock-*D*istributed *Q*ueue, *T*icket *L*ock, *C*ache opt.)

Der faire Ticket-Lock wird in dieser Implementierung zur Sicherstellung des wechselseitigen Ausschlusses verwendet.

DQ-BL-TL-TRY-C (*BL*ock-*D*istributed *Q*ueue, *T*icket *L*ock, only single *TRY*, *C*ache opt.)

Wie in der vorigen Implementierung *DQ-BL-TL-C* wird der Ticket-Lock verwendet, allerdings in einer semi nicht blockierenden Variante. Dabei wird ein Ticket nur dann mit Einsatz der Hardware-Operationen bestimmt, wenn der Lock zum Zeitpunkt des Tests frei ist. Ist der Lock belegt, wird beim nächsten Thread in der Suchreihenfolge nach verfügbaren Tasks gesucht.

DQ-BL-SP-C (*BL*ock-*D*istributed *Q*ueue, simple lock with *S*ingle*P*ut extension, *C*ache opt.)

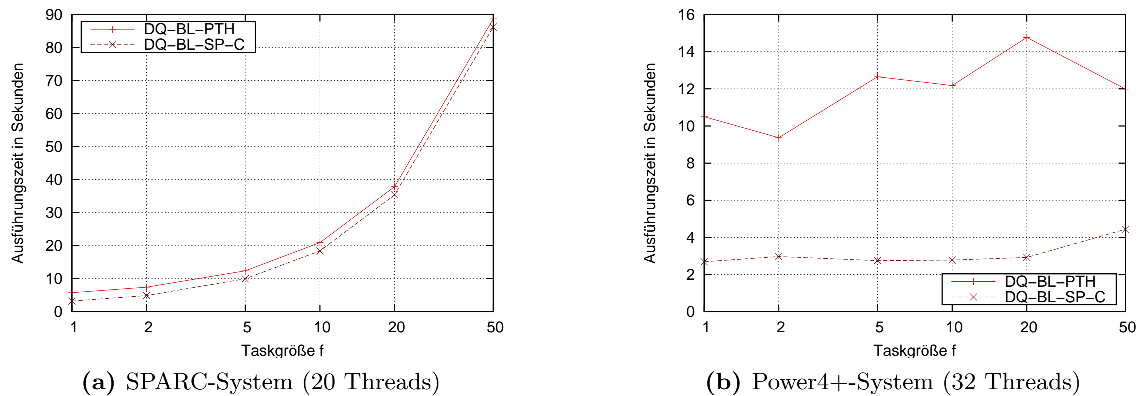
Diese Implementierung setzt die Erweiterung der Einfügeoperation ein, womit das Einfügen eines Blocks in die öffentliche Liste ohne Synchronisationsoperationen durchgeführt werden kann.

DQ-BL-PTH (*BL*ock-*D*istributed *Q*ueue, *PTH*reads)

Die Vergleichsimplementierung implementiert die gleiche Gruppierung von vier Tasks pro Block wie bei den vorherigen blockbasierten Taskpools und kontrolliert den Zugriff auf die verteilten Listen mit Pthreads-Mutex-Locks.

Performance-Analyse

Die Laufzeitexperimente mit den blockverteilten Taskpools zeigen, dass sich bei der synthetischen Applikation die Performance wesentlich verbessert, ohne eine ungleichmäßige



(a) SPARC-System (20 Threads)

(b) Power4+-System (32 Threads)

Abbildung 4.15: Ausführungszeit der synthetischen Applikation mit blockverteilten Taskpools für unterschiedliche Taskgrößen (parallele Ausführung).

Lastverteilung durch die Taskgruppierung zu erzeugen. In Abbildung 4.14a sind die Messergebnisse für einen Thread auf dem SPARC-System dargestellt. Das Laufzeitverhalten ist nahezu linear in Abhängigkeit von der Taskgröße, wobei die Unterschiede der Hardware-optimierten Varianten gegenüber der Pthreads-Implementierung so klein sind, dass hier nur eine Variante abgebildet ist. Auf der schnelleren Power4+-Architektur (Abbildung 4.14b) ist das Ergebnis nicht wesentlich anders, allerdings ist der Unterschied speziell für sehr kleine Tasks noch zu erkennen.

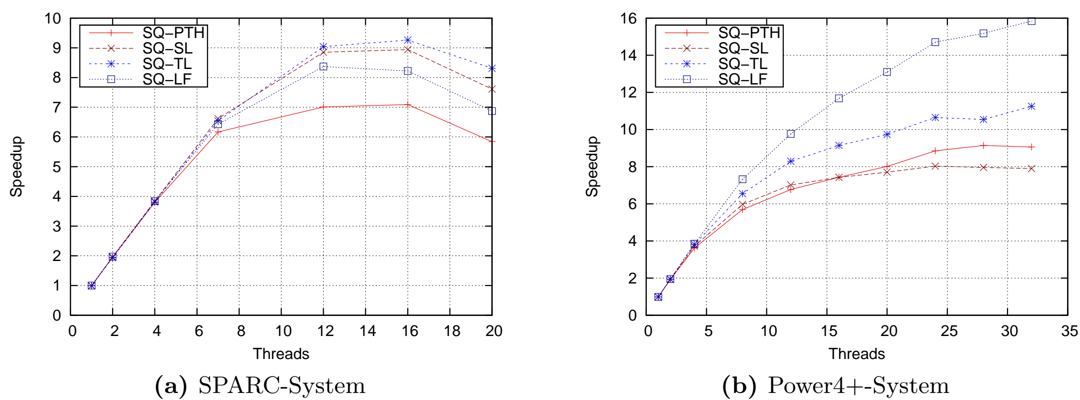
Bei der parallelen Ausführung zeigen sich größere Unterschiede. Auf der SPARC-Architektur (Abbildung 4.15a) mit 20 Threads wird eine nahezu ideale Skalierbarkeit erreicht. Allerdings ist der prozentuale Unterschied der Laufzeiten der Hardware-unterstützten Taskpools zu der Pthreads-Implementierung größer, besonders für feingranulare Tasks.

Auf dem Power4+-System ist der Unterschied wie schon bei den verteilten Taskpools ohne Gruppierung größer. Bei 32 Threads (Abbildung 4.15b) erzielen die Hardware-unterstützten Implementierungen wesentlich kürzere Laufzeiten als die Pthreads-Implementierung, die bis zu einem Faktor von $5,06$ langsamer ist. Der Overhead der Taskverwaltung ist bei den Hardware-unterstützten Taskpools nur noch für sehr kleine Tasks relevant, schon ab einer Taskgröße von $f \geq 20$ kann die zu erwartende lineare Abhängigkeit von dem Faktor f gemessen werden. Dabei erreicht die Implementierung mit der modifizierten Einfügeoperation ohne Synchronisation die besten Ergebnisse. Die schlechtere Datenlokalität wird hier durch die Taskgruppierung gemildert.

In Tabelle 4.4 sind die Ergebnisse der verschiedenen Implementierungen für eine Taskgröße von $f = 50$ zusammengefasst. Auf dem SPARC-System wird der maximale Speedup von $19,81$ von der Implementierung mit der modifizierten Einfügeoperation erreicht, was einer Effizienz von $0,99$ entspricht. Die Pthreads-Implementierung ist dagegen etwas langsamer und erreicht nur eine Effizienz von $0,96$. Auf dem Power4+-System dagegen erreicht die Pthreads-Implementierung bei 32 Threads nur einen Speedup von $10,07$ mit einer schlechten Effizienz von $0,31$ wogegen die beste Implementierung mit Hardware-Operationen einen Speedup von $27,18$ erreicht, was einer Effizienz von $0,85$ entspricht.

Taskpool-Implementierung	SPARC			Power4+		
	Höchster Speedup	Anzahl Threads	Effizienz	Höchster Speedup	Anzahl Threads	Effizienz
DQ-BL-PTH	19,26	20	0,96	10,07	32	0,31
DQ-BL-SL-C	19,77	20	0,989	26,82	32	0,84
DQ-BL-TL-C	19,76	20	0,988	26,70	32	0,83
DQ-BL-SP-C	**19,81**	20	**0,99**	**27,18**	32	**0,85**

Tabelle 4.4: Speedup und Effizienz der blockverteilten Taskpools mit einer Taskgröße $f = 50$ auf dem SPARC- und dem Power4+-System.



(a) SPARC-System

(b) Power4+-System

Abbildung 4.16: Speedups der hierarchischen Radiosity-Applikation für die Szene „largeroom“ mit zentralen Taskpools.

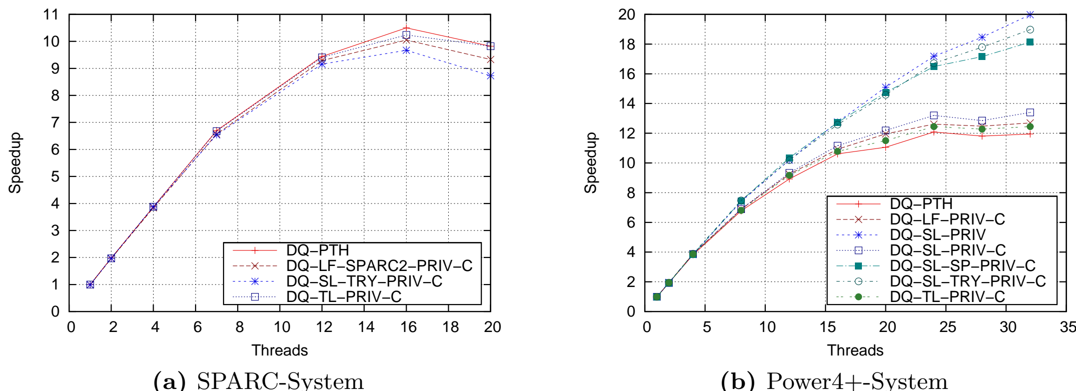
4.4 Laufzeitergebnisse mit realistischen Applikationen

Im Folgenden sollen realistische Applikationen betrachtet werden, um die erzielbaren Verbesserungen der Hardware-unterstützten Taskpools zu messen. Dazu werden Applikationen aus Kapitel 3 untersucht.

4.4.1 Hierarchisches Radiosity

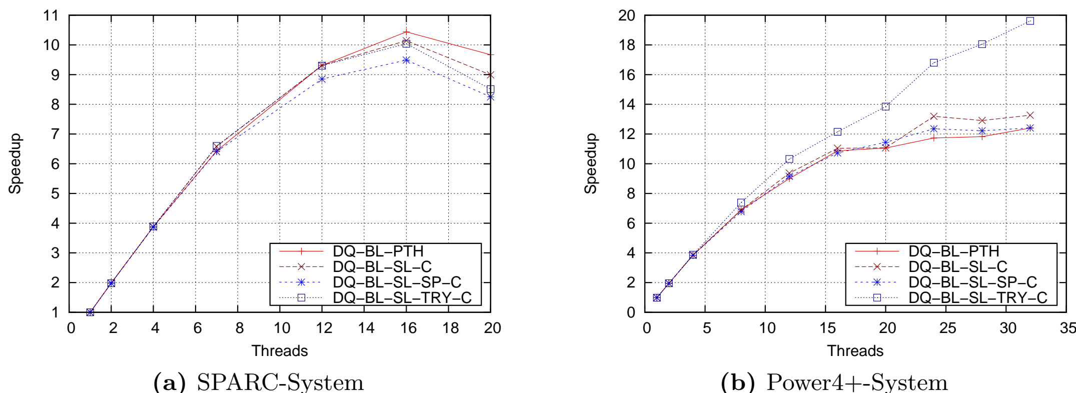
Das hierarchische Radiosity stellt durch die komplexe Taskstruktur und die dynamische Taskerzeugung hohe Anforderungen an die Taskpool-Implementierung. Als Basis der Messungen wird die Szene „largeroom“ verwendet. Der erreichte Speedup wird in Bezug zu einer sequenziellen Taskpool-Implementierung ohne Synchronisationsoperationen berechnet.

Abbildung 4.16 zeigt die Ergebnisse der Messungen mit den zentralen Taskpools. Auf der SPARC-Architektur (Abbildung 4.16a) erzielt die Taskpool-Implementierung mit Ticket-Locks mit 9,26 den höchsten Speedup während die Pthreads-Implementierung nur einen Speedup von 7,09 erreicht. Während die sperrfreie Implementierung auf der SPARC-Architektur langsamer als z. B. die Ticket-Lock-Implementierung ist, so erreicht die äquivalente Implementierung auf dem Power4+-System (Abbildung 4.16b) mit 15,84 den höchsten



(a) SPARC-System **(b)** Power4+-System

Abbildung 4.17: Speedups der hierarchischen Radiosity-Applikation für die Szene „largeroom“ mit verteilten Taskpools.



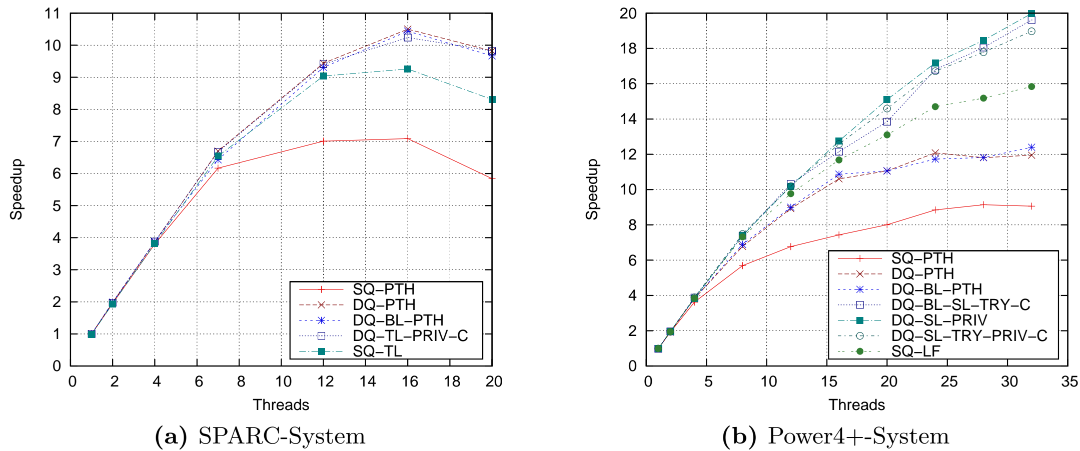
(a) SPARC-System **(b)** Power4+-System

Abbildung 4.18: Speedups der hierarchischen Radiosity-Applikation für die Szene „largeroom“ mit blockverteilten Taskpools.

Speedup. Dagegen erzielt die Pthreads-Implementierung nur einen Speedup von 9, 14.

Der Einsatz verteilter Taskpools kann den Speedup weiter erhöhen (Abbildung 4.17). Auf der SPARC-Architektur erreicht die Pthreads-Implementierung einen Speedup von 10, 5 (Abbildung 4.17a). Allerdings sind alle auf Hardware-Operationen basierenden Implementierungen etwas langsamer als die Pthreads-Implementierung. Dies weist auf ein Problem innerhalb der Applikation hin, das im weiteren Verlauf der Arbeit noch genauer untersucht wird. Bei den Hardware-basierten Taskpools treten Skalierbarkeitsprobleme in der Applikation durch den geringeren Overhead eher zu Tage. Auf dem Power4+-System (Abbildung 4.17b) erreichen die verteilten Taskpools etwas bessere Laufzeiten, allerdings wird auch hier nicht der optimale Speedup erreicht. Die Pthreads-Implementierung erreicht einen Speedup von 12, 08 während die beste Implementierung *DQ-SL-PRIV* einen Speedup von 19, 98 erreicht.

Die blockverteilten Taskpools (Abbildung 4.18) können wie erwartet keine wesentlichen



(a) SPARC-System

(b) Power4+-System

Abbildung 4.19: Zusammenfassung der Speedups der hierarchischen Radiosity-Applikation für die Szene „largeroom“.

Verbesserungen mehr erzielen, da die Skalierbarkeit innerhalb der Applikation begrenzt ist. Auf der SPARC-Architektur (Abbildung 4.18a) bleibt die Pthreads-Implementierung auf gleichem Niveau wie die Implementierung mit verteilten Tasklisten ohne Gruppierung. Auf dem Power4+-System ist die Pthreads-Implementierung mit einem Speedup von 12,4 zwar etwas schneller als die vergleichbare Implementierung ohne Gruppierung. Die beste Hardware-basierte Implementierung erreicht dagegen einen Speedup von 19,61, der somit etwas unter dem besten Speedup der verteilten Taskpools ohne Gruppierung liegt.

In Abbildung 4.19 sind nochmal zur besseren Vergleichbarkeit alle Ergebnisse der verschiedenen Taskpool-Implementierungen zusammengefasst. Zentrale Taskpools stellen bei dieser Applikation einen Flaschenhals dar. Auf beiden Architekturen erreichen diese Taskpools wesentlich schlechtere Speedups. Allerdings kann schon der Einsatz von Hardware-Operationen die Geschwindigkeit wesentlich verbessern. Auf dem SPARC-System (Abbildung 4.19a) ist die zentrale Implementierung mit Ticket-Locks (SQ-TL) deutlich schneller als der vergleichbare zentrale Taskpool mit Pthreads-Locks (SQ-PTH) und kommt den Speedups der verteilten Taskpools relativ nah. Auf der Power4+-Architektur (Abbildung 4.19b) erreicht der zentrale Taskpool mit sperrfreier Listenimplementierung mittels Hardware-Operationen (SQ-LF) sogar einen höheren Speedup als verteilte Taskpools mit Pthreads-Synchronisationsoperationen (z. B. DQ-BL-PTH). Durch die Ersetzung der zentralen Liste durch verteilte Listen und der Benutzung von Hardware-Operationen zur Synchronisation konnte die Geschwindigkeit mehr als verdoppelt werden.

4.4.2 Paralleles Quicksort

Das parallele Quicksort-Verfahren stellt durch seine Taskstruktur spezielle Anforderungen an die Taskpool-Implementierung. Da anfangs nur wenige Tasks verfügbar sind, sollten möglichst frühzeitig viele dieser Tasks an verschiedene Threads zugewiesen werden, um einen guten Speedup zu erreichen.

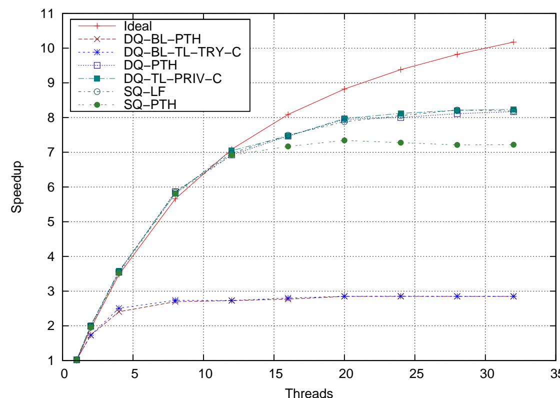


Abbildung 4.20: Speedup der parallelen Quicksort-Implementierung auf dem Power4+-System mit einer Feldgröße von 100.000.000.

Abbildung 4.20 zeigt die Speedups für das Power4+-System bei einer zu sortierenden Feldgröße von 100 Millionen Zahlen. Zusätzlich ist der im Idealfall zu erreichende Speedup entsprechend der Abbildung 3.3 aus Kapitel 3.5 abgebildet. Es ist direkt zu sehen, dass der ideale Speedup bei einer größeren Anzahl benutzter Threads nicht erreicht werden kann, da viele Parameter, wie insbesondere eine mögliche ungleiche Feldaufteilung, in der Berechnung des idealen Speedups nicht berücksichtigt werden.

Darüber hinaus können im Wesentlichen zwei Gruppen identifiziert werden:

1. Die blockverteilten Taskpools erzielen nur einen maximalen Speedup von weniger als drei. Zwar reduziert die Taskgruppierung den Overhead bei dem Task-Stealing und der Verwaltung der Tasks, allerdings hindert dies bei dieser Applikation andere Threads an der Abarbeitung ausführungsbereiter Tasks. Die Tasks einer nicht vollständig gefüllten Gruppe werden im privaten Teil des Taskpools gespeichert und können von keinem anderen Thread ausgeführt werden. Da gerade am Anfang nur wenige Tasks existieren, ist nur ein einzelner Thread an der Ausführung beteiligt.

2. Die zentralen und verteilten Taskpools ohne Gruppierung können die verfügbare Parallelität besser ausnutzen. Für 8 Threads werden sogar leicht höhere Speedups erreicht als durch den idealen Speedup mit $5,66$ vorgegeben, was auf die Nichtberücksichtigung der besseren Cache-Ausnutzung durch mehrere Prozessoren zurückzuführen ist.

   Aus dieser Gruppe tritt mit einer größeren Threadanzahl nur der zentrale Taskpool mit Pthreads-Synchronisationsoperationen heraus, der nur einen Speedup von $7,34$ bei 20 Threads erreicht. Die anderen Implementierungen erreichen dagegen ähnliche Speedups. Die Pthreads-basierte verteilte Taskpool-Implementierung erreicht einen Speedup von $8,17$ bei 32 Threads während die beste Implementierung mit Hardware-Operationen basierend auf dem Ticket-Lock einen Speedup von $8,23$ erreicht. Das Ergebnis des sperrfreien Taskpools bestätigt die gute Performance auf dieser Plattform. Selbst der sperrfreie zentrale Taskpool erreicht mit einem Speedup von $8,22$ eine bessere Performance als der verteilte Taskpool mit Pthreads-Operationen.

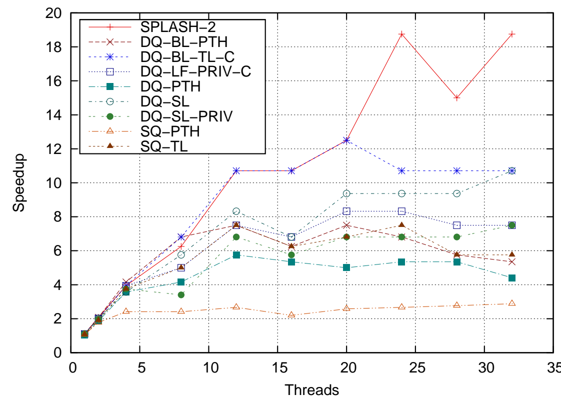


Abbildung 4.21: Speedup der adaptiven Volume-Rendering-Applikation für die Szene „head“ auf dem Power4+-System.

Bei dieser Applikation ist also die Verfügbarkeit ausführungsbereiter Tasks wesentlich wichtiger für eine gute Performance als die Wahl der Taskpool-Implementierung. Dennoch kann durch Nutzung effizienter Synchronisationsmechanismen selbst ein einfacher zentraler Ansatz verfolgt werden, um gute Resultate zu erzielen.

4.4.3 Volume-Rendering

Die parallele Implementierung des Volume-Rendering-Verfahrens erzeugt während der Taskabarbeitung keine neuen Tasks, sodass hier im Wesentlichen nur der Overhead der Taskentnahme und des Task-Stealings eine Rolle spielt. Die zur Verfügung stehenden Dichtedaten als $256 \times 256 \times 128$-Datenblock zur Darstellung eines 2D-Bildes sind für aktuelle Rechnersystem relativ klein, sodass die Berechnung eines Bildes nur wenig Zeit in Anspruch nimmt.

In Abbildung 4.21 sind die Ergebnisse der Messungen mit der Szene „head“ auf dem Power4+-System dargestellt. Dabei wird die adaptive Berechnung des Volume-Rendering-Verfahrens benutzt.

Zusätzlich zu den Taskpool-basierten Implementierungen ist auch der Speedup der originalen SPLASH-2-Implementierung abgebildet, die eine speziell auf die Applikation zugeschnittene Lastbalancierung verwendet. Diese erreicht bei diesem Test mit einem Speedup von 18, 75 auch die beste Performance. Die beste Taskpool-Implementierung nutzt die blockverteilte Taskabspeicherung mit Ticket-Locks und erreicht bei 20 Threads einen Speedup von 12, 5 und ist dort noch genauso schnell wie die SPLASH-2-Implementierung. Im direkten Vergleich sind allerdings die Hardware-basierten Taskpool-Implementierungen bis zu 66% schneller als die Pthreads-basierten Implementierungen. Auch erzielt wiederholt der zentrale sperrfreie Taskpool bessere Laufzeiten als verteilte Taskpools mit Pthreads-Locks. Die einfachste Variante mit einer zentralen Liste und Pthreads-Locks erreicht nur einen Speedup von ungefähr 3.

Die schlechte Performance der Taskpool-Implementierungen im Vergleich zur SPLASH-2-Implementierung für eine größere Anzahl von Threads liegt in der applikationsspezifischen

Implementierung	Speedup
SPLASH-2	26,1
SQ-PTH	6,7
SQ-TL	21,7
DQ-PTH	13,6
DQ-LF-PRIV-C	29,1
DQ-BL-PTH	18,2
DQ-BL-SP-C	29,1

Tabelle 4.5: Speedups der nicht adaptiven Volume-Rendering-Applikation für die Szene „head“ auf dem Power4+-System.

Lastbalancierung der SPLASH-2-Implementierung. Durch die geringe Taskgranularität wird besonders häufig auf den Taskpool zugegriffen. Da während der Berechnung keine neuen Tasks erzeugt werden, kann insbesondere wiederkehrendes Task-Stealing ein Problem darstellen. In der SPLASH-2-Implementierung wird direkt ein einfacher Zähler für jeden Thread benutzt, der den nächsten zu berechnenden Bildteil adressiert. Während für jeden Task im Taskpool entsprechende Listen durchsucht werden müssen, reicht in der Applikation ein einfaches Erhöhen des Zählers, der durch einen Pthreads-Lock geschützt ist. Einen solch geringen Overhead kann der Taskpool aufgrund seines generischen Aufbaus nicht erzielen. Dennoch ist zu erkennen, dass in einem solchen Fall die blockverteilten Taskpools gute Ergebnisse erzielen können und für eine geringere Anzahl von Threads mindestens genauso schnell sind wie das applikationsspezifische Lastbalancierungsverfahren.

Die geringe Größe der Volume-Rendering-Tasks ist ein Grund für die relativ geringe Performance. Das nicht adaptive Volume-Rendering erreicht dagegen bessere Werte (Tabelle 4.5). Während hier die originale SPLASH-2-Implementierung einen Speedup von $26,1$ erreicht, kann der beste Hardware-basierte Taskpool jetzt einen Speedup von $29,1$ erreichen. Auch hier erzielen die Hardware-unterstützten Taskpools bessere Ergebnisse als die vergleichbaren Taskpools mit Pthreads-Locks.

4.4.4 Ray-Tracing

Bei der Ray-Tracing-Applikation werden ähnlich wie bei der Volume-Rendering-Applikation keine Tasks während der Abarbeitung neu erzeugt, allerdings sind die mit einem Task verbundenen Berechnungen aufwendiger.

Abbildung 4.22 zeigt die Speedups auf dem Power4+-System mit der Szene „car“ bei einer Bildauflösung von 512×512 Pixeln.

Alle Taskpool-Implementierungen erreichen wesentliche bessere Speedups als die originale SPLASH-2-Implementierung, die nur einen Speedup von $4,45$ bei 12 Threads erreicht. Die Skalierbarkeit ist dabei durch zentrale Lock-geschützte Listen beschränkt, die die originale Implementierung für die eigene taskbasierte Abarbeitung nutzt. Da dies für die Taskpools nicht notwendig ist bzw. die entsprechende Funktionalität durch den Taskpool bereitgestellt wird, können diese Implementierungen bessere Resultate erzielen. Dabei erreicht schon die langsamste Implementierung mit einer zentralen Liste und Pthreads-Locks einen Speedup von $24,92$. Die beste Implementierung nutzt den Hardware-unterstützten Ticket-Lock mit

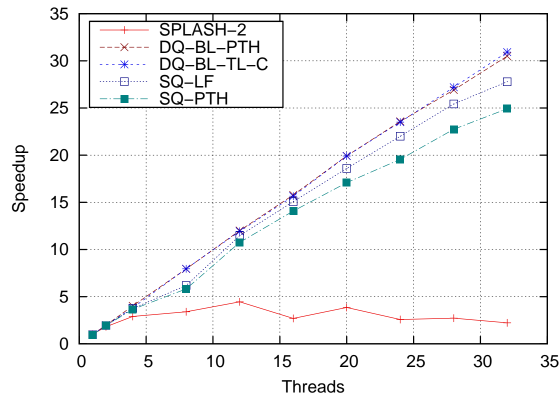


Abbildung 4.22: Speedup der Ray-Tracing-Applikation auf dem Power4+-System (Szene „car“, 512×512).

blockverteilten Tasklisten und erreicht bei 32 Thread einen nahezu idealen Speedup von $30,92$. Allerdings ist der vergleichbare Pthreads-basierte blockverteilte Taskpool mit einem Speedup von $30,5$ nur geringfügig langsamer.

4.5 Zusammenfassung

Die Benutzung von Hardware-Operationen zur Realisierung der Synchronisation zwischen Threads kann die Performance taskbasierter Programme erheblich beschleunigen. Tabelle 4.6 stellt die relevanten Ergebnisse der betrachteten Applikationen dar.

Aus der Anzahl der verarbeiteten Tasks und deren Ausführungszeit lässt sich eine durchschnittliche Taskgröße bestimmen, die ein gutes Bewertungskriterium dafür ist, wie gut eine Applikation für eine taskbasierte Ausführung geeignet ist. Bei Tasks, die nur eine geringe Ausführungszeit haben, sind die erzielten Speedups geringer. So hat das adaptive Volume-Rendering eine durchschnittliche Taskgröße von nur etwa 2 Mikrosekunden. Der Overhead der Taskverwaltung fällt entsprechend stark ins Gewicht und der beste Speedup der Taskpools ist nur $12,5$. Bei der nicht adaptiven Variante steigt die Taskgröße auf 6 Mikrosekunden, wodurch sich der Speedup schon wesentlich erhöht. Die Ray-Tracing-Applikation hat mit 420 Mikrosekunden durchschnittlicher Taskgröße mehr als 200 mal größere Tasks als die adaptive Volume-Rendering-Applikation und erreicht praktisch idealen Speedup. Aufgrund der großen Tasks ist der Einfluss der verschiedenen Taskpool-Implementierungen geringer, sodass selbst mit Pthreads-Locks gute Ergebnisse erzielt werden können. Die Taskgröße des hierarchischen Radiosity-Verfahrens ist mit 84 Mikrosekunden zwar kleiner als die Taskgröße des Ray-Tracing-Verfahrens, allerdings wesentlich größer als die Tasks der nicht adaptiven Volume-Rendering-Applikation. Dennoch kann kein idealer Speedup erzielt werden. Allerdings ist diese Applikation durch dynamische Taskerzeugung mit verschiedenen Tasktypen und Berechnungsphasen wesentlich komplexer. Daher wird die Radiosity-Applikation im weiteren Verlauf der Arbeit eine wichtige Rolle bei der Analyse verschiedener Lastbalancierungsverfahren spielen.

Die parallele Quicksort-Applikation stellt einen Sonderfall dar, der ebenfalls interessant

Applikation	Volume-Rendering (adaptiv) Szene *head*	Volume-Rendering (nicht adaptiv) Szene *head*	Ray-Tracing Szene *car*	Radiosity Szene *largeroom*	Quicksort Feldgröße 100.000.000
Ausführungszeit	75 ms	204 ms	6876 ms	14931 ms	16502 ms
Anzahl Tasks	35344	35344	16384	177881	150894
Durchschnittliche Taskgröße	$\approx 2\,\mu s$	$\approx 6\,\mu s$	$\approx 420\,\mu s$	$\approx 84\,\mu s$	$\approx 109\,\mu s$
Maximaler Speedup SPLASH-2	**18,75**	26,12	4,45	11,42	N/A
Maximaler Speedup Pthreads-basierter Taskpools	7,50	18,19	30,50	12,40	8,17
Maximaler Speedup Hardware-basierter Taskpools	12,50	**29,11**	**30,92**	**19,98**	**8,23**

Tabelle 4.6: Speedups und Taskgrößen der betrachteten Applikationen.

für die taskbasierte Programmabarbeitung ist. Die durchschnittliche Taskgröße ist zwar etwas größer gegenüber z. B. der Radiosity-Applikation, allerdings versteckt die Durchschnittsberechnung die Charakteristik der Tasks, deren Größe mit jeder Berechnungsstufe etwa halbiert werden. Somit existieren anfangs wenige große, am Ende sehr viele sehr kleine Tasks mit entsprechend unterschiedlichen Anforderungen an die Taskpool-Implementierung. Die zentralen und verteilten Taskpools ohne Gruppierung können mit dieser Taskstruktur gut umgehen, während die blockverteilten Taskpools keine hohen Speedups erreichen können.

Kapitel 5

Implementierung des KOALA-Frameworks

In den beiden vorherigen Kapiteln wurden taskbasierte Applikationen und deren Ausführung mit Hardware-unterstützten Taskpools untersucht. Hier sollen nun die Implementierungen der einzelnen Komponenten des in Kapitel 2 vorgestellten KOALA-Frameworks erläutert werden. Dabei fließen die Erfahrungen aus Kapitel 4 in den Entwurf effizienter Komponenten mit ein, sodass die Synchronisationsoperationen effizient innerhalb des Frameworks genutzt werden können. Durch Nutzung der entsprechenden Komponenten können im Unterschied zu den Implementierungen im vorherigen Kapitel auch die Applikationen direkt die verbesserten Synchronisationsoperationen nutzen. Da jedoch aus Sicht der Applikationen nur die jeweilige Komponentenschnittstelle genutzt wird, erfordert eine Änderung in der Synchronisationskomponente keine Änderungen der jeweiligen Applikation.

5.1 Speichermanager

Im Folgenden werden die verfügbaren Implementierungen der Speichermanager-Komponente beschrieben. Die jeweilige Implementierung muss das in Kapitel 2.3 beschriebene Interface implementieren. Der Speichermanager muss Blöcke fest definierter Größe für die Applikation zur Nutzung als Taskargument bereitstellen und zusätzlich Blöcke ebenfalls fest definierter Größe zur Speicherung von Tasks für das Taskpool-Backend bereitstellen. Die betrachteten Implementierungen unterscheiden sich in der Ausführungshäufigkeit globaler Betriebssystem-Allokationsfunktionen und der Speicherausnutzung zur Verringerung von Cache-Fehlzugriffen.

Die zu nutzende Komponente mit der jeweiligen Bezeichnung „<Name>“ kann mit der *configure*-Option *--enable-mem=<Name>* beim Übersetzen aktiviert werden. Ohne Angabe wird die Implementierung mit den meisten Optimierungen „memopt“ verwendet, die allerdings auch mehr Speicherplatz benötigt als andere Varianten. Falls dies also mit einer bestimmten Applikation bzw. deren Eingabedaten problematisch wird, kann zu einer anderen Implementierung gegriffen werden, ohne Änderungen an der Applikation vornehmen zu müssen.

5.1.1 Speichermanager „membs“

Als Basisimplementierung wird eine Variante verwendet, die die Aufrufe der Interface-Funktionen direkt an die entsprechenden Funktionen der Standard-C-Bibliothek zur Speicherallokation und -freigabe weiterreicht. Dazu rufen die Funktionen *tp_alloc_arg* und

tp_alloc_task direkt die Funktion *malloc* mit den entsprechenden Task- bzw. Argumentgrößen auf, wie sie bei der Initialisierung angegeben wurden. Analog findet bei den Freigabefunktionen *tp_free_arg* und *tp_free_task* jeweils ein Aufruf von *free* statt, um die übergebenen Speicherelemente wieder freizugeben.

Diese Speichermanager-Variante wird nur für Test- und Vergleichszwecke verwendet, da sie mit dem größten Overhead verbunden ist. Andererseits können so auch externe Speichermanager genutzt werden, die *malloc* und *free* durch eigene Implementierungen ersetzen.

5.1.2 Blockbasierter Speichermanager „memdgbk“

Bei früheren Untersuchungen [58, 69] hat sich für den Speichermanager ein blockbasierter Ansatz mit Freilisten zur Zwischenspeicherung freier Blöcke als guter Kompromiss verschiedener Ansätze herausgestellt. Die Implementierung *memdgbk* (verteilte, wachsende Liste von Blöcken) setzt diesen Ansatz als Komponente im Framework um.

Wesentlicher Punkt ist eine synchronisationsfreie Behandlung der Speicherblöcke. Dazu erhält jeder Thread eine eigene Datenstruktur, mit denen seine lokalen Speicherblöcke verwaltet werden.

Jeder von der Applikationsschicht oder der Backend-Schicht freigegebene Speicherblock wird in einer der jeweiligen Schicht zugehörigen Thread-lokalen Liste gespeichert. Aus dieser Liste werden zukünftige Anfragen für den jeweiligen Thread bedient. Somit werden Speicherblöcke vor Programmende nicht wieder freigegeben, sondern wiederverwendet.

Stehen keine freien Blöcke in der Thread-lokalen Freiliste zur Verfügung, muss Speicherplatz vom Betriebssystem allokiert werden. Dazu werden Verwaltungsblöcke mit *malloc* allokiert, deren Größe ein Vielfaches eines Speicherblocks der Applikations- oder Backend-Schicht betragen. Wenn bei einer Anfrage die zu dem Thread und der Schicht gehörende Freiliste keine Blöcke enthält, wird aus diesem Verwaltungsblock ein neuer Speicherblock entnommen und zurückgegeben. Bei einer späteren Freigabe wird der Speicherblock dann in der Freiliste gespeichert. Sobald der Verwaltungsblock komplett in Speicherblöcke zerlegt wurde, wird ein nächster Verwaltungsblock allokiert.

Für ein spezielles Programm gibt die maximale Anzahl der zu einem beliebigem Zeitpunkt gleichzeitig existierenden, ausführungsbereiten Tasks somit den maximalen Bedarf an Speicherblöcken an. Unabhängig von der Anzahl der insgesamt ausgeführten Tasks wird nicht mehr Speicher benötigt, da alle sonstigen Speicheranfragen aus der jeweiligen Freiliste bedient werden können.

5.1.3 Statischer Speichermanager „memstatic“

Um den Einfluss der Speichermanager-Implementierung auf die restlichen Komponenten zu minimieren, existiert die *memstatic*-Variante. Ähnlich wie die vorher beschriebene Implementierung *memdgbk* verwendet auch dieser Speichermanager Freilisten, um freigegebene Speicherblöcke wiederzuverwenden. Allerdings wird als Reservoir für neue Speicherblöcke nur einmal zu Beginn der Ausführung ein großer Verwaltungsblock allokiert, aus dem alle Speicheranfragen bedient werden müssen. Der Speichermanager muss also im Voraus wissen, wie viele Tasks bei der betrachteten Applikation gleichzeitig existieren können. Der

Verwaltungsblock muss also durch vorheriges Festlegen einer festen Größe genügend Platz für alle nötigen Speicherblöcke bieten.

Jeder Thread erhält einen solchen Verwaltungsblock, der zudem komplett initialisiert wird. In modernen Betriebssystemen wird Speicher bei einer Anforderung zugeteilt, ohne dass tatsächlich Systemspeicher dafür reserviert wird. Erst bei der ersten Benutzung wird die virtuelle Speicherseite auf eine physikalische Seite abgebildet, wodurch eventuell weitere Operationen seitens des Betriebssystems ausgelöst werden. So muss z. B. durch „Paging“ eventuell eine andere Speicherseite ausgelagert werden. Solche Einflüsse sind durch vorhergehende Initialisierung reduziert.

Diese Implementierung dient hauptsächlich Vergleichszwecken, da die feste Grenze für die Verwaltungsblockgröße in der Praxis oft nicht bekannt ist und so oft wesentlich mehr Speicher allokiert werden muss als tatsächlich benötigt wird.

5.1.4 Zugriffsoptimierter Speichermanager „memopt“

Die *memopt*-Implementierung der Speichermanager-Komponente ist eine Implementierung des blockbasierten Ansatzes, um Speicherfehlzugriffe zu reduzieren. Dieser Speichermanager ist die voreingestellte Speichermanagerkomponente und wird beim Übersetzen automatisch ausgewählt, solange keine andere Variante angegeben ist. Besonderer Wert wurde auf die Verhinderung von „False-Sharing“ bei der Benutzung der Speicherblöcke innerhalb der Applikation und des Taskpool-Backends gelegt.

False-Sharing beschreibt allgemein die Abbildung von mindestens zwei logisch verschiedenen Systemressourcen auf die gleiche (physikalische) Systemressource.

Die in modernen Systemen vorhandene Speicherhierarchie mit Level1-, Level2- und eventuell Level3-Cache und dem eigentlichen Hauptspeicher benutzt als Adressierungseinheit nicht die Speicheradresse, sondern je nach Hardware und Konfiguration unterschiedlich zusammengefasste Speicherbereiche.

Innerhalb der Cache-Hierarchie beschreiben Cache-Zeilen (*cache lines*) zusammenhängende Speicheradressen, die jeweils immer gemeinsam als Block geladen oder ausgelagert werden. Typische Größen von Cache-Zeilen sind derzeit 32 oder 64 Byte (die sogenannte *cache line size*), die also jeweils in einem Stück aus dem Hauptspeicher in den Cache übertragen werden. Für Auslagerungen von Speicherblöcken auf externe Datenträger wie Festplatten werden darüber hinaus Cache-Zeilen zu Speicherseiten, sogenannte *Pages*, zusammengefasst, die einige wenige Kilobyte, aber auch mehrere Megabyte umfassen können (genannt *page size*).

Zwei benachbarte Speicheradressen liegen also oft in der gleichen Cache-Zeile. Bei einem Zugriff auf die erste Adresse sorgt das Speichersystem dafür, dass die entsprechende Cache-Zeile in den Cache transportiert wird. Dies kann aus einem der darunterliegenden Caches (Level 2 oder Level 3) oder aus dem Hauptspeicher erfolgen. Der Zugriff auf die zweite Adresse erfolgt dann ohne Verzögerung, da die gemeinsame Cache-Zeile schon verfügbar ist. Diese *räumliche Lokalität* (benachbarter Zugriff) ist neben der *zeitlichen Lokalität* (zeitlich wiederholender Zugriff auf die gleiche Adresse) ein gewünschter Effekt der Cache-Hierarchie.

Bei der Benutzung mehrerer Threads in einem System mit gemeinsamem Speicher kann das *False-Sharing* die Performance erheblich reduzieren. Greifen z. B. zwei Prozessoren auf

aus ihrer Sicht getrennte Daten zu, die aber in der gleichen Cache-Zeile liegen, so liegt die gleiche Cache-Zeile dupliziert in den Caches der beiden Prozessoren. Führt einer dieser Prozessoren nun einen Schreibzugriff auf sein Datum aus, so muss das Speichersystem die Cache-Zeile im Cache des anderen Prozessors als ungültig kennzeichnen, um Speicherkonsistenz zu gewährleisten. Ein erneuter Lesezugriff des betroffenen Prozessors wird nun verzögert, da erst die modifizierte Cache-Zeile aus dem Hauptspeicher (oder dem Cache des anderen Prozessors) gelesen werden muss.

Dies kann an mehreren Stellen des Taskpool-Frameworks zu Performance-Problemen führen. Unmittelbar betroffen sind die internen Datenstrukturen für die Speicherung ausführungsbereiter Tasks. Im Laufe der Abarbeitung werden zwecks Lastbalancierung Tasks zwischen Prozessoren verschoben, sodass Speicherblöcke von Prozessor A durch Prozessor B benutzt werden. Dies passiert z. B., wenn Prozessor A einen Task erzeugt, der von Prozessor B jedoch entfernt und ausgeführt wird, wenn dieser keine eigenen Tasks mehr zur Verfügung hat. Bei der Freigabe der Datenstruktur wird dann der eigentlich von Prozessor A stammende Speicherblock in die Freiliste des Prozessors B eingetragen und somit später wiederverwendet. Insbesondere bei feingranularen Tasks führt dies zu einer hohen Anzahl von False-Sharing-Zugriffen.

Auch die Applikationsschicht kann von diesem Problem betroffen sein, da die Taskargumente der erzeugten Tasks von dem Speichermanager verwaltet werden. Greift die Applikation bei der Ausführung eines Tasks besonders häufig auf die Taskargumente zu, so können auch hier mehr Cache-Fehlzugriffe auftreten, wenn der Task von einem anderen Prozessor erzeugt wurde.

Die Speichermanager-Implementierung *memopt* vermeidet solche Fehlzugriffe durch optimierte Speicherbenutzung anhand von systemabhängigen Parametern. Dazu werden die Datenstrukturen so im Speicher abgelegt, dass keine zwei Instanzen einer Datenstruktur in einer gemeinsamen Cache-Zeile liegen können. Dies ist jedoch nur durch Einfügen zusätzlicher Füllbytes möglich, die ungenutzt bleiben.

Wie in Kapitel 2.3.1 erwähnt gibt der Programmierer an, wie viele Bytes ein Taskargument in Anspruch nimmt (*arg_size*). In ähnlicher Weise gibt die Taskpool-Backend-Implementierung vor, wie viele Bytes eine Taskdatenstruktur umfasst (*taskstruct_size*). Die Werte dieser beiden Variablen werden gemäß Formel 5.1 erhöht, damit die Größe der Speicherblöcke ein Vielfaches der Cache-Zeilengröße betragen.

$$\text{padded_size}(size) = \lfloor \frac{(size + \text{pad_size} - 1)}{\text{pad_size}} \rfloor \cdot \text{pad_size} \tag{5.1}$$

Die Variable *size* beschreibt also den von der Applikation benötigten Speicherplatz oder die Task-Strukturgröße für das Taskpool-Backend. Das Resultat der Funktion *padded_size()* ist ein Vielfaches der Variable *pad_size* und wird nur intern verwendet.

Die Wahl des Parameters *pad_size* hängt vom tatsächlichen System ab. Zwar ist die Cache-Zeilengröße meist bekannt, kann aber nicht als absolutes Kriterium zum Ausschluss von False-Sharing benutzt werden. In modernen Speichersystemen werden auch darüber hinaus größere Speicherblöcke für Zugriffsoptimierungen betrachtet, indem z. B. durch *Prefetching*-Mechanismen mehr als eine Cache-Zeile nachgeladen werden. In Kapitel 5.4 wird die Wahl dieses Parameters genauer betrachtet.

```
struct tp_item_t {
    struct tp_item_t *next;
};

struct tp_mem_t {
  tp_item_t *arg_buf;
  tp_item_t *task_buf;

  unsigned long arg_index;
  unsigned long task_index;

  tp_item_t *free_args;
  tp_item_t *free_tasks;

  void *next_arg;
  void *next_task;
};

tp_mem_t tp_mem[];
```

Abbildung 5.1: Datenstruktur zur Speicherung von Speicherelementen eines Threads.

Größere Werte von *pad_size* verringern das False-Sharing, erfordern aber mehr Speicher. Um den dadurch entstehenden erhöhten Speicherbedarf einzugrenzen, verwendet die Speichermanager-Implementierung *memopt* zwei unterschiedliche Konstanten für die Ausrichtung der Speicherblöcke an den Cache-Zeilengrenzen.

Ist der erhöhte Speicherverbrauch durch die Füllbytes nicht kritisch, wird die Konstante **TP_COARSE_PADDING** benutzt. Der Wert wird so gewählt, dass eine Störung zwischen Prozessoren nahezu ausgeschlossen ist. Typischerweise wird dazu die Seitengröße benutzt (*page_size*), also in Abhängigkeit des Systems z. B. 4096 Byte. Dies wird z. B. für die Ablage der internen Datenstrukturen für jeden Thread benutzt, da die Anzahl der Threads im Verhältnis zur Speichergröße klein und von dem zu bearbeitenden Problem unabhängig ist.

Für die Taskargumente und die internen Taskstrukturen wird die Konstante **TP_FINE_PADDING** benutzt, deren Wert typischerweise wesentlich kleiner als der Wert für **TP_COARSE_PADDING** ist und sich an der tatsächlichen Cache-Zeilengröße orientiert. Für die Ermittlung des geeigneten Wertes wurde eine Tuning-Komponente entworfen, die in Kapitel 5.4 genauer beschrieben ist und für die jeweils betrachtete Systemarchitektur einen optimalen Wert bestimmt.

Die Implementierung des Speichermanagers legt für jeden Thread eine Instanz der in Abbildung 5.1 gezeigten Datenstruktur derart im Speicher ab, dass der Abstand zwischen benachbarten Datenstrukturen jeweils **TP_COARSE_PADDING** Bytes umfasst und so Wechselwirkungen ausschließt.

Die Struktur **tp_item_t** als einfach verkettete Liste wird als Basisstruktur für alle Speicherelemente benutzt und gibt somit die Mindestgröße des jeweiligen Speicherblocks an.

Die eigentliche Speicherstruktur **tp_mem_t** speichert in den Elementen *arg_buf* und *task_buf* für jeden Thread eine Liste von allokierten Verwaltungsblöcken, aus denen sukzessive Taskargument- oder Taskstrukturelemente entnommen werden. Der Index *arg_index*

bzw. *task_index* gibt jeweils den nächsten freien Eintrag innerhalb des Verwaltungsblocks an.

Die schon erwähnte Freiliste wird in den beiden Struktureinträgen *free_args* und *free_tasks* verwaltet. Zusätzlich gibt es noch jeweils einen direkten Zeiger auf ein einzelnes freies Element (*next_arg* und *next_task*), das zur schnelleren Bearbeitung der häufigen Situation der unmittelbaren Freigabe eines Elements und der anschließenden Anfrage eines neuen Elements benutzt wird.

Dem Problem des False-Sharing wird Rechnung getragen, indem die Feldelemente für die einzelnen Threads weit genug voneinander entfernt sind. Der Ort der Speicherung der Datenstrukturen für Thread i wird entsprechend um den Faktor **TP_COARSE_PADDING** verschoben, sodass das jeweilige Feld an der Position

$$tp_mem[\mathrm{TP_COARSE_PADDING} \cdot i]$$

abgespeichert ist.

Um die Aufrufe der Funktion *malloc* zu reduzieren, werden die benötigten Elemente blockweise allokiert und gemäß der dazugehörigen Variablen *arg_index* bzw. *task_index* entnommen. Bestimmt durch die Konstante **TP_MEMBUF_SIZE** werden entsprechend viele Speicherblöcke im Verwaltungsblock abgelegt. Dessen Größe beträgt daher

$$\begin{aligned} &size_of\,(\mathrm{tp_item_t}) + \\ &TP_MEMBUF_SIZE \cdot \mathrm{padded_size}\,(size) \end{aligned}$$

Bytes, wobei *size* entweder *arg_size* für die Taskargumentblöcke oder *task_size* für die internen Taskblöcke ist. Die einzelnen Elemente eines Verwaltungsblocks können zwar durch Task-Stealing in den Freilisten eines anderen Threads nach Beendigung des Tasks abgespeichert werden, durch die Anpassung der Strukturgröße mit padded_size() ist der Einfluss von False-Sharing aber weitestgehend reduziert.

5.2 Lock-Komponente

Die einzusetzende Implementierung der Lock-Komponente kann mit der *configure*-Option *--enable-lock=<Name>* aktiviert werden. Dabei wird die Verfügbarkeit auf dem Zielsystem überprüft, sodass nur tatsächlich unterstützte Lock-Mechanismen ausgewählt werden können. Die jeweiligen Komponenten implementieren das in Abschnitt 2.4.1 beschriebene Interface, sodass beim Austausch der Implementierung keine Programmänderung nötig ist.

5.2.1 Pthreads-Lock „mutexlock“

Die Pthreads-Bibliothek bietet neben der Threadverwaltung auch Funktionen zur Threadsynchronisation. Mit den *Pthreads-Mutex-Locks* kann die Ausführung eines kritischen Bereiches sequenzialisiert werden. Dazu fordert ein Thread den zugehörigen Lock an, indem er die Funktion *pthread_mutex_lock()* ausführt. Die Freigabe erfolgt durch Aufruf von *pthread_mutex_unlock()*.

```
struct tplock_t {
    pthread_mutex_t lock;
};

int tplock_init( tplock_t *l )
{
    return pthread_mutex_init( &( lock->lock ), NULL );
}

static inline int tplock_lock  ( tplock_t *l )
{
    return pthread_mutex_lock( &( l->lock ) );
}

static inline int tplock_unlock( tplock_t *l )
{
    return pthread_mutex_unlock( &( l->lock ) );
}
```

Abbildung 5.2: Realisierung der Pthreads-Lock-basierten Lock-Komponente.

Um eine hohe Portierbarkeit des Taskpool-Frameworks zu gewährleisten, benutzt diese Implementierung direkt die Pthreads-Mutex-Locks wie in Abbildung 5.2 dargestellt.

Die Datenstruktur *tplock_t* speichert nur die entsprechende Pthreads-Datenstruktur, sodass nicht mehr Speicher verbraucht wird als bei direkter Pthreads-Nutzung. Die Funktionen zur Sperrung und Freigabe des Locks *tplock_lock()* bzw. *tplock_unlock()* sind als *inline*-Funktionen deklariert und stellen somit bei der Nutzung optimierender Compiler keinen Funktionsoverhead dar.

Da diese Implementierung auf allen Systemen, die Pthreads unterstützen, verfügbar ist und weder einen Speicher- noch einen Rechenzeit-Overhead mitbringt, können Pthreads-Locks innerhalb einer Applikation direkt durch das Taskpool-Lock-Interface ersetzt werden.

5.2.2 Pthreads-Spinlock „spinlock“

Die Pthreads-Bibliothek stellt neben dem Mutex-Lock einen weiteren Lock bereit, der allerdings nur optional von einer Pthreads-Implementierung unterstützt werden muss. Daher prüft das Taskpool-Framework zur Compile-Zeit, ob dieser Lock unterstützt wird und macht diese Komponente gegebenenfalls verfügbar.

Der Spinlock verzichtet auf eine Prozesssuspendierung für den Fall, dass der Lock gesperrt ist. Stattdessen wird wiederholt versucht, den Lock zu erhalten („busy loop“). Da der Thread nicht schlafen gelegt wird und entsprechend bei einer Lock-Freigabe durch einen anderen Thread nicht durch Signal-Mechanismen aufgeweckt werden muss, entfällt ein wesentlicher Overhead beim Aufruf der Betriebssystemfunktionen.

Die Implementierung dieser Komponente erfolgt analog zur Pthreads-Mutex-Lock-Komponente wie in Abbildung 5.3 dargestellt. Die Funktionsaufrufe von *tplock_lock()* und *tplock_unlock()* werden direkt durch *pthread_spin_lock()* bzw. *pthread_spin_unlock()* ersetzt.

```
struct tplock_t {
    pthread_spinlock_t lock;
};

int tplock_init( tplock_t *l )
{
    return pthread_spin_init( &( lock->lock ), 0 );
}

static inline int tplock_lock   ( tplock_t *l )
{
    return pthread_spin_lock( &( l->lock ) );
}

static inline int tplock_unlock( tplock_t *l )
{
    return pthread_spin_unlock( &( l->lock ) );
}
```

Abbildung 5.3: Realisierung der Pthreads-Spinlock-basierten Lock-Komponente.

5.2.3 Hardware-Lock „hwlock“

Die Untersuchungen in Kapitel 4 zeigten ein großes Potenzial bei der Benutzung von Hardware-unterstützten Synchronisationsoperationen. Da die Lock-Komponente allerdings nur Sperrmechanismen abstrahiert, können sperrfreie Konzepte hier nicht integriert werden. Dennoch kann die direkte Nutzung der Hardware-Operationen im Quellcode der Applikation zu Laufzeitverbesserungen führen. Da die Hardware-Lock-Komponente im Gegensatz zur Pthreads-Lock-Komponente plattformabhängig ist, existieren mehrere Implementierungen. Im Laufe der Entwicklungsarbeit des KOALA-Frameworks wurde das SGI-Altix-4700-System am LRZ München verfügbar, das über wesentlich mehr Prozessoren mit gemeinsamem Speicher verfügt und so ein interessantes Untersuchungsobjekt darstellt. Derzeit sind daher Hardware-Locks für die Plattformen „IA-64“ (Intel Itanium Prozessoren) und „x86“ (Intel ix86 Prozessoren und kompatible (AMD,...)) als eigene Komponente implementiert. Andere Architekturen lassen sich jedoch einfach integrieren. Da die Lock-Operationen möglichst ohne Funktionsoverhead in den benutzenden Code eingebettet werden sollen, müssen entsprechende Mechanismen der Compiler genutzt werden, direkt Hardware-Operationen in Form der Assembler-Anweisungen einzufügen. Dies fügt zur Plattformabhängigkeit noch die Abhängigkeit vom genutzten Compiler hinzu, die aber ebenfalls, soweit möglich, automatisch aufgelöst wird. Das Konfigurationsskript versucht den Typ des Compilers zu erkennen und entsprechend nutzbare Codeblöcke zu aktivieren. Dabei werden die auf einer Plattform relevanten Compiler berücksichtigt. Dies ist auf allen Plattformen die *GNU Compiler Collection*, die mit der *asm*-Anweisung das Einbetten von Assembler-Code in normalen C-Code erlaubt. Dieser Compiler wird auf den verfügbaren Plattformen sehr gut unterstützt, allerdings erzielen die speziellen Compiler der Systemhersteller oft bessere Laufzeitresultate. Auf der „x86“-Architektur wird daher zusätzlich der *Intel C Compiler (ICC)* unterstützt, der auf dieser Plattform die *asm*-Anweisung ebenso unterstützt. Auf der Itanium-Plattform „IA-64“ wird derzeit die *asm*-Anweisung vom *ICC* nicht unterstützt, sodass auf andere Mechanismen zurückgegriffen werden muss, die im folgenden Abschnitt betrachtet werden.

```
struct tplock_t {
    uint64_t lock;
};

int tplock_init( tplock_t *l )
{
    lock->lock = 0;
    return 0;
}
```

Abbildung 5.4: Verwendete Datenstruktur für den Hardware-Lock der IA-64-Architektur.

Architektur IA-64

Die von Intel entwickelte Architektur IA-64 wurde in den Itanium-Prozessoren implementiert und stellt besondere Anforderungen an die Programmierung. Die theoretisch hohe Peak-Performance wird nur durch geeignete Programmierung erreicht. Die IA-64-Architektur erlaubt die explizite Spezifizierung der parallel auszuführenden Maschinenbefehle, wodurch dem Compiler mehr Möglichkeiten zur Optimierung gegeben werden. Andererseits kann eine hohe Performance aber auch nur durch Ausnutzung der parallelen Instruktionsslots erreicht werden. Obwohl der Itanium-Prozessor auch Instruktionen der IA-32-Architektur ausführen kann, ist der reguläre IA-64-Betriebsmodus dazu inkompatibel.

Mit der SGI Altix 4700 existiert am LRZ München eine große Maschine mit Itanium2-Prozessoren und gemeinsamem Adressraum, die sich gut für die Untersuchungen eignet. Daher wurde eine Komponente für Hardware-Locks auf Basis der IA-64-Architektur implementiert.

Die Implementierung nutzt die in Abbildung 5.4 dargestellte Struktur zur Realisierung des Locks. Der Lock wird nur durch den Zustand Null für *frei* und Eins für *belegt* beschrieben. Da die Speicherzugriffe für 64-Bit optimiert sind, wird dennoch eine 64-Bit-Variable für diesen Lock-Zustand verwendet.

Die Umsetzung der Lock- bzw. Unlock-Operation in Abbildung 5.5 folgt im Wesentlichen der Beispielimplementierung im *Architecture Manual* von Intel [61]. Dabei wird auf den *Compare & Swap* -Mechanismus zurückgegriffen, wie er in Kapitel 4.2 beschrieben wurde. Der atomare Austausch findet in Zeile 7 statt und Zeile 8-9 prüft das erfolgreiche Austauschen der Variable mit dem Wert Eins (gesperrt). Der Vorgang wird entsprechend sooft wiederholt, bis der Lock erfolgreich gesperrt worden ist. Die Freigabe erfolgt durch Zurückschreiben des Wertes Null in Zeile 18.

Der Intel-C-Compiler (ICC) unterstützt auf der IA-64-Architektur die *asm*-Anweisung derzeit nicht, sodass für diesen Compiler auf eine alternative Implementierung zurückgegriffen wird. Der Compiler stellt stattdessen sogenannte „Intrinsics“ bereit, die die Assemblerbefehle kapseln und so als normale Funktionen im C-Code verwendet werden können. Abbildung 5.6 zeigt die entsprechende Implementierung. Im Wesentlichen wird auch hier die Schleife solange wiederholt, bis die Anweisung *CMPXCHG* erfolgreich ausgeführt und der Lock gesperrt werden konnte. Die Freigabe erfolgt durch Zurücksetzen der Sperrvariable auf den Wert Null. Der Intel-Compiler wird die Intrinsics-Anweisungen *ia64_barrier* und *ia64_cmpxchg8* durch die äquivalenten Assemblerbefehle ersetzen und so ähnlichen Code

```
1  static inline int tplock_lock  ( tplock_t *l )
2  {
3      uint64_t temp1, temp2;
4      asm volatile( "        mov ar.ccv = 0"
5                    "        mov %0 = 1;;"
6                    "1:"
7                    "        cmpxchg8.acq %1 =[%2], %0;;"
8                    "        cmp.eq p10, p0 = %1, %0"
9                    "(p10) br.cond.spnt 1b;;"
10                   : "=&r" (temp1), "=&r" (temp2)
11                   : "r" (&l->lock)
12                   : "memory", "ar.ccv", "p10" );
13     return 0;
14 }
15
16 static inline int tplock_unlock( tplock_t *l )
17 {
18     asm volatile( "        st8.rel [%0] = r0;;"
19                   :
20                   : "r" (&l->lock)
21                   : "memory" );
22     return 0;
23 }
```

Abbildung 5.5: Realisierung des Hardware-Locks auf der IA-64-Architektur.

```
static inline int tplock_lock  ( tplock_t *l )
{
    uint64_t temp;

    ia64_barrier();
    do {
        temp = ia64_cmpxchg8_acq( &l->lock, 1, 0 );
        ia64_barrier();
    } while ( temp != 0 );
    return 0;
}

static inline int tplock_unlock( tplock_t *l )
{
    ia64_barrier();
    ( (volatile tplock_t *)l )->lock = 0;
    return 0;
}
```

Abbildung 5.6: Alternative Realisierung des Hardware-Locks auf der IA-64-Architektur.

```
struct tplock_t {
    uint64_t lock;
};

int tplock_init( tplock_t *l )
{
    lock->lock = 1;
    return 0;
}
```

Abbildung 5.7: Verwendete Datenstruktur für den Hardware-Lock der x86-Architektur.

erzeugen wie in Abbildung 5.5. Aus Gründen der Lesbarkeit wurde auf die Intrinsics-Makros des Linux-Kernels zurückgegriffen, die auf die tatsächlichen Funktionen des ICC abgebildet werden.

Architektur x86

Zu weiteren Testzwecken wurde der Hardware-Lock auf der Intel-x86-Architektur (ia-32) sowohl in der 32-Bit- als auch in der 64-Bit-Variante (x86_64 oder AMD64, EM64T bzw. Intel 64) implementiert. Abbildung 5.7 zeigt die dafür verwendete Datenstruktur. Da alle aktuellen Prozessoren der x86-Architektur von Intel und AMD die 64-Bit-Erweiterung unterstützen, wurde vorrangig die 64-Bit-Umgebung betrachtet, wobei der Code jedoch auch für eine 32-Bit-Zielplattform genutzt werden kann. Während im *Intel Architecture Manual* [62] für diese Plattform die Benutzung der Assembleranweisung *XCHG* vorgeschlagen wird, wird hier stattdessen auf die Implementierung im Linux-Kernel zurückgegriffen, die mit weniger Operationen den Lock effizienter realisiert (siehe *include/asm-x86_64/spinlock.h* in den Quellen des Linux-Kernels).

Anstelle einer einfachen Null/Eins-Implementierung wird eine 64-Bit-Variable als Zähler verwendet. Der Wert Eins zeigt einen freien Lock an und wird somit als Initialwert verwendet. Wie in Abbildung 5.8 zu sehen, wird dieser Zähler vom anfragenden Thread atomar dekrementiert (Zeile 4). Da der eigentliche Wert durch diese Operation nicht verfügbar ist, wird das Vorzeichen des Resultats geprüft. Nur dem Thread, der den Zähler von 1 auf 0 reduziert hat, wird der Lock zugeteilt. Alle anderen nachfolgenden Versuche reduzieren den Zähler in den negativen Zahlenbereich, sodass direkt auf die Freigabe gewartet werden kann. Der Lock-Eigentümer gibt den Lock durch Zurückschreiben des Wertes Eins (Zeile 19) frei.

5.2.4 Hardware-Ticket-Lock „hwticketlock“

Der Ticket-Lock stellt einen fairen Lock dar, bei dem die verhältnismäßig teure Synchronisationsoperation nur einmalig ausgeführt werden muss und somit weniger Belastung des Speichersystems auftritt. In Kapitel 4 konnten entsprechende gute Resultate erzielt werden. Daher wurde auch eine Implementierung der Lock-Komponente realisiert, die auf diesem Lock aufbaut.

Für die beiden Architekturen „IA-64“ und „x86“ wird die in Abbildung 5.9 dargestellte Datenstruktur verwendet. Die Variable „next_ticket“ gibt dabei das nächste freie Ticket an, welches durch anfragende Threads mit entsprechenden Synchronisationsoperationen erhöht

```
1  static inline int tplock_lock  ( tplock_t *l )
2  {
3      asm volatile( "1:"
4                    "    lock decl %0"
5                    "    jns 3f"
6                    "2:"
7                    "    cmpl $1, %0"
8                    "    jne 2b"
9                    "    jmp 1b"
10                   "3:"
11                   : "=m" (l->lock)
12                   :
13                   : "memory" );
14     return 0;
15 }
16
17 static inline int tplock_unlock( tplock_t *l )
18 {
19     asm volatile( "movl $1, %0"
20                   : "=m" (l->lock)
21                   :
22                   : "memory" );
23     return 0;
24 }
```

Abbildung 5.8: Realisierung des Hardware-Locks auf der x86-Architektur.

```
struct tplock_t {
    uint32_t next_ticket;
    uint32_t serve_ticket;
};

int tplock_init( tplock_t *l )
{
    lock->next_ticket = lock->serve_ticket = 0;
    return 0;
}
```

Abbildung 5.9: Verwendete Datenstruktur für den Hardware-Ticket-Lock.

werden muss. Die zweite Variable „serve_ticket" gibt das aktuelle Ticket an, das Zutritt zum kritischen Bereich erhält. Als Startsituation werden also beide mit dem gleichen Wert initialisiert.

Architektur IA-64

Der Befehlssatz der IA-64-Architektur enthält den atomaren Befehl „fetchadd", mit dem direkt der Ticket-Lock implementiert werden kann, wie dies in Abbildung 5.10 für den GNU-Compiler dargestellt ist. Da dieser Befehl immer erfolgreich ist, ist keine Schleife notwendig, um den atomaren Befehl zu wiederholen (Zeile 4). Die eigentliche Warteschleife (Zeile 9-11) greift also nur lesend auf die Variable „serve_ticket" zu, die somit solange direkt aus dem Cache des entsprechenden Prozessors gelesen wird, bis der Lock durch Erhöhung der Variable freigegeben wird.

```
 1  static inline int tplock_lock  ( tplock_t *l )
 2  {
 3      uint32_t ticket;
 4      asm volatile( "fetchadd4.acq %0=[%1],1"
 5                    : "=&r" (ticket)
 6                    : "r" (&l->next_ticket)
 7                    : "memory" );
 8
 9      while ( ticket !=
10              ((volatile tplock_t*)l)->serve_ticket )
11          asm_barrier();
12      return 0;
13  }
14
15  static inline int tplock_unlock( tplock_t *l )
16  {
17      uint32_t ticket;
18      asm volatile( "fetchadd4.rel %0=[%1],1"
19                    : "=&r" (ticket)
20                    : "r" (&l->serve_ticket)
21                    : "memory" );
22      return 0;
23  }
```

Abbildung 5.10: Realisierung des Hardware-Ticket-Locks auf der IA-64-Architektur mit dem GNU-Compiler.

```
static inline int tplock_lock  ( tplock_t *l )
{
    uint32_t ticket;

    ia64_barrier();
    ticket = ia64_fetchadd4_acq( &l->next_ticket, 1 );

    while ( ticket !=
            ((volatile tplock_t*)l)->serve_ticket )
       ia64_barrier();
    return 0;
}

static inline int tplock_unlock( tplock_t *l )
{
    ia64_barrier();
    ia64_fetchadd4_rel( &l->serve_ticket, 1 );
    return 0;
}
```

Abbildung 5.11: Alternative Realisierung des Hardware-Ticket-Locks auf der IA-64-Architektur mit dem Intel-Compiler.

```
1  static inline int tplock_lock  ( tplock_t *l )
2  {
3      uint32_t temp;
4      asm volatile( "1:"
5                    "     movl  %1, %%eax"
6                    "     movl  %%eax, %0"
7                    "     incl  %0"
8                    "lock cmpxchgl %0,%1"
9                    "     jnz  1b"
10                   "2:"
11                   "     cmpl  %%eax, %2"
12                   "     jnz  2b"
13                   : "=&r" (temp),
14                     "=m" (l->next_ticket),
15                     "=m" ( l->serve_ticket )
16                   :
17                   : "eax", "memory" );
18     return 0;
19 }
20
21 static inline int tplock_unlock( tplock_t *l )
22 {
23     asm volatile( "incl %0"
24                   : "=m" (l->serve_ticket)
25                   :
26                   : "memory" );
27     return 0;
28 }
```

Abbildung 5.12: Realisierung des Hardware-Ticket-Locks auf der x86-Architektur (GCC- und ICC-Compiler).

Abbildung 5.11 zeigt die Implementierung bei der Benutzung des Intel-C-Compilers. Hier wird auf die Intrinsics-Anweisung *ia64_fetchadd4* zurückgegriffen, um ein Ticket zu erhalten, während die Wartephase identisch implementiert ist.

Architektur x86

Auf der x86-Architektur gibt es den Befehl *fetchadd* nicht, der atomar den aktuellen Wert liest, erhöht und sowohl im Hauptspeicher als auch in einem Register ablegt. Daher wird wie in Abbildung 5.12 dargestellt das *Compare & Swap*-Verfahren in Form des Befehls *CMPXCHG* benutzt, um den zuvor gelesenen und erhöhten Wert (Zeile 6-7) atomar zurückzuschreiben, wenn er nicht zwischenzeitlich verändert wurde (Zeile 8-9).

Der Aufwand für den Ticket-Lock ist auf dieser Architektur also höher als für die IA-64-Architektur, da im Konfliktfall die atomare Operation wiederholt ausgeführt werden muss. Dies ist beim einfachen Hardware-Lock aber genauso nötig, sodass demgegenüber kein zusätzlicher Overhead entsteht.

5.2.5 Duolock

Der Duolock teilt dem Eigentümer des Locks ein Vorrangsrecht zu, um insbesondere bei verteilten Datenstrukturen effizientere Modifikationen für den Eigentümer zu erlauben. Dazu wird ein zweistufiges Lock-System benutzt, sodass ein Thread priorisiert Zugriff auf den

```
struct tp_duolock_t {
    int owner_flag, other_flag;
    char padding[TP_FINE_PADDING];
    tplock_t other_lock;
};

int tp_dl_init( tp_duolock_t *lock )
{
    lock->owner_flag = 0;
    lock->other_flag = 0;
    tplock_init( &lock->other_lock );
    return 0;
}
```

Abbildung 5.13: Verwendete Datenstruktur für den Duolock.

```
static inline int tp_dl_owner_lock( tp_duolock_t *l )
{
    volatile tp_duolock_t *rl = l;
    for(;;) {
        rl->owner_flag = 1;
        hw_mem_barrier();
        if ( rl->other_flag == 0 ) return 0;

        rl->owner_flag = 0;
        while ( rl->other_flag == 1 );
    }
}

static inline int tp_dl_owner_unlock( tp_duolock_t *l )
{
    hw_mem_barrier();
    ((volatile tp_duolock_t*)l)->owner_flag = 0;
    return 0;
}
```

Abbildung 5.14: Realisierung des Duolocks aus Sicht des Lock-Eigentümers.

Lock erlaubt wird, indem der zu einem Duolock zugeordnete Thread nur mit jeweils einem Thread der externen Gruppe von Threads um die Zuteilung konkurrieren muss. Wie in Kapitel 2.4.2 beschrieben, existieren dafür zwei verschiedene Funktionsklassen, die zuerst den Lock-Eigentümer mit nur einem anderen Thread synchronisieren und in einer zweiten Stufe alle anderen Threads untereinander sequenzialisieren.

Bei der hier verwendeten Implementierung auf Basis von Dekkers Algorithmus [32] wird die Datenstruktur aus Abbildung 5.13 verwendet. Im Wesentlichen werden zwei Variablen eingesetzt, wobei ein gesetzter Wert Eins anzeigt, dass der jeweilige Partner den Lock sperren will. Zusätzlich existiert der äußere Lock *other_lock*, mit dem alle anderen Threads untereinander synchronisiert werden, um einzeln mit dem Lock-Eigentümer um den eigentlichen Lock zu konkurrieren.

Abbildung 5.14 zeigt dazu die Implementierung aus Sicht des Lock-Eigentümers. Zuerst wird die entsprechende Variable des Eigentümers auf Eins gesetzt, um den eventuell anderen Thread zu signalisieren, dass der Lock gesperrt werden soll. Es folgt eine Speicherbarriere,

```
 1  static inline int tp_dl_other_lock( tp_duolock_t *l )
 2  {
 3      volatile tp_duolock_t *rl = l;
 4
 5      tplock_lock( &l->other_lock );
 6
 7      for(;;) {
 8          rl->other_flag = 1;
 9          hw_mem_barrier();
10          if ( rl->owner_flag == 0 ) return 0;
11
12          rl->other_flag = 0;
13          while ( rl->owner_flag == 1 );
14      }
15  }
16
17  static inline int tp_dl_other_unlock( tp_duolock_t *l )
18  {
19      hw_mem_barrier();
20      ((volatile tp_duolock_t*)l)->other_flag = 0;
21      return tplock_unlock( &l->other_lock );
22  }
```

Abbildung 5.15: Realisierung des Duolocks aus Sicht der externen Threads.

womit sichergestellt wird, dass die Variable im Speicher tatsächlich den Wert Eins besitzt. Auf diese Barrierenfunktion wird im Abschnitt 5.3.1 genauer eingegangen. Anschließend wird überprüft, ob der andere Thread ebenfalls den Lock sperren wollte. Ist dies nicht der Fall, ist der Lock an den Eigentümer zugeteilt und die Funktion wird verlassen. Wenn jedoch ein anderer Thread in der Zwischenzeit auch den Lock angefragt oder schon gesperrt hat, setzt der Thread die eigene Variable wieder zurück und wartet auf die Freigabe. Danach wird der gesamte Vorgang wiederholt. Es kann dabei ein wechselseitiges Blockieren auftreten, wenn beide Threads immer gleichzeitig ihre Variable setzen. Wenn diese Störung signifikant ist, kann eine unterschiedlich lange Wartezeit eingefügt werden, die eine zeitliche Verschiebung gewährleistet. Die Freigabe des gesperrten Locks erfolgt durch einfaches Zurücksetzen der entsprechenden Variable.

Die externen Threads, die nicht als Eigentümer des Locks betrachtet werden, müssen vor der Sperrung des inneren Locks zuvor untereinander synchronisiert werden. Abbildung 5.15 zeigt dazu das entsprechende Vorgehen. Vor Zugriff auf den inneren Zwei-Stufen-Lock wird die Lock-Variable *other_lock* mit der normalen Lock-Komponente gesperrt (Zeile 5). Somit kann nur jeweils ein Thread den inneren Lock anfragen, wobei das Programm identisch zu der Eigentümervariante ist, nur mit umgekehrter Nutzung der Variablen *owner_flag* und *other_flag*. Die Freigabe erfolgt durch Freigabe des inneren Locks, indem die Variable *other_flag* wieder auf Null gesetzt wird. Danach wird der äußere Lock freigegeben, sodass ein anderer Thread der externen Gruppe nun den inneren Lock anfragen kann.

Die Unterscheidung zwischen Lock-Eigentümer und den anderen Threads erfolgt aus Effizienzgründen nicht automatisch, sondern muss durch den Nutzer dieser Komponente explizit durch die Verwendung der jeweiligen Funktionen angegeben werden. Nur ein Thread darf die Funktionen *tp_dl_owner_lock* und *tp_dl_owner_unlock* des Lock-Eigentümers aufrufen. Die Funktionen *tp_dl_other_lock* und *tp_dl_other_unlock* können dagegen von

```
static inline void hw_code_barrier();
static inline void hw_mem_barrier();
```

Abbildung 5.16: Interface der Hardware-Barrier-Unterstützung.

beliebig vielen Threads verwendet werden, allerdings ist hier natürlich der Lock-Eigentümer ausgeschlossen.

Darüber hinaus bleibt die Implementierung des Duolocks vor dem Anwender verborgen. Obwohl der Lock eine nahezu reine Software-Lösung ist, so besteht doch durch die verwendete Speicherbarriere eine Abhängigkeit zur jeweiligen Plattform. Wenn der Duolock auf der Zielplattform nicht implementiert werden kann, weil eine entsprechende Implementierung dieser Barriere nicht vorliegt, dann kann der Duolock mit dem *configure*-Argument „--disable-duolock“ deaktiviert werden, sodass ein normaler Lock verwendet wird. Bei der Applikation, die diesen Duolock nutzt, sind jedoch keine Änderungen nötig.

5.3 Unterstützende Funktionen

Das KOALA-Framework bietet wie in Kapitel 2.5 beschrieben einige zusätzliche Funktionen, mit denen einerseits von Hardware-abhängigen Details abstrahiert wird und andererseits in der parallelen Programmierung häufig verwendete grundlegende Funktionalitäten bereitgestellt wird. Im Folgenden werden die einzelnen Implementierungen dargestellt.

5.3.1 Code- und Speicherbarrieren

Moderne Compiler versuchen häufig, den Programmcode durch Transformationen so zu optimieren, dass er möglichst effizient ausgeführt werden kann [5]. Dabei kann es zu Umordnungen des Programmcodes kommen, die bei einer sequenziellen Ausführung korrekt sind, aber im Zusammenspiel mit anderen Threads zu einer inkorrekten Programmausführung führen. Darüber hinaus ordnen moderne Prozessoren die auszuführenden Instruktionen im Rahmen ihrer Möglichkeiten gegebenenfalls auch um, damit die Funktionseinheiten besser ausgenutzt werden können (*out-of-order execution* [52]). Zudem können Lese- und Schreiboperationen verzögert oder gruppiert ausgeführt werden, um den Speicher besser nutzen zu können. Auch dies kann bei zeitkritischem parallelen Code zu einer inkorrekten Ausführung führen.

Bei der Implementierung paralleler Algorithmen gibt es jedoch häufig zeitkritische Bereiche, deren Abarbeitung nach genau festgelegten Regeln erfolgen muss, um korrekt zu funktionieren. Greifen z. B. mehrere Threads auf gemeinsame Variablen zu, kann die Reihenfolge erheblichen Einfluss auf die Korrektheit haben.

Daher bietet das KOALA-Framework die in Abbildung 5.16 dargestellten Funktionen zur Sicherstellung einer korrekten Abarbeitungsreihenfolge. Die (Makro-)Funktion *hw_code_-barrier* stellt sicher, dass der Compiler keine Anweisungen vor der Barriere in den Bereich nach der Barriere verschiebt und entsprechend keine nachfolgenden Anweisungen vorzieht. Da die Implementierung von der genutzten Compiler-Version abhängt, existieren verschiedene Varianten, wovon die richtige automatisch während des Compile-Schrittes ausgewählt

```
GCC und XLC:
  static inline void hw_code_barrier()
  {
    asm volatile( "" : : : "memory" );
  }

ICC:
  static inline void hw_code_barrier()
  {
    ia64_barrier();
  }
```

Abbildung 5.17: Implementierung der Code-Barrier-Unterstützung für verschiedene Compiler.

wird. Abbildung 5.17 zeigt die verfügbaren Implementierungen für die Compiler GCC, XLC (von IBM für Power-Prozessoren) und ICC (Intel). Der GCC- und der XLC-Compiler unterstützen die *asm*-Anweisung, sodass die leere Anweisung mit dem *memory*-Merkmal den Compiler veranlasst, über diese Anweisung keine Optimierungen durchzuführen. Der ICC unterstützt für die IA-64-Plattform die *asm*-Anweisung nicht, daher wird die Intrinsics-Anweisung *ia64_barrier* benutzt, die auf den Compiler die gleiche Auswirkung hat.

Durch die *out-of-order*-Ausführung von Anweisungen und andere prozessorinterne Optimierungen können Speicheroperationen in einer anderen Reihenfolge ausgeführt werden als im Quellcode vorgegeben. Die zweite Funktion *hw_mem_barrier* ist eine Hardware-abhängige Funktion zur Synchronisation von Speicheroperationen. Diese verhindert ein Umordnen von Speicherzugriffen auf Prozessorebene und stellt sicher, dass entsprechende Zugriffe zu dem festgelegten Zeitpunkt tatsächlich ausgeführt und im Hauptspeicher angekommen sind. Für die verschiedenen Prozessoren existieren unterschiedliche Anweisungen. So ist es teilweise möglich, nur Lese- oder Schreiboperationen zu synchronisieren. Um die verschiedenen Operationen zu vereinheitlichen, wird hier nur eine Barriere realisiert, die sämtliche Speicheroperationen an diesem Punkt synchronisiert. Abbildung 5.18 zeigt die vorhandenen Implementierungen für die betrachteten Plattformen. Im Wesentlichen wird nur eine Instruktion ausgeführt, mit der aber eine erhebliche Wartezeit verbunden sein kann, wenn Lese- oder Schreiboperationen beendet werden müssen.

Diese wenig komplexe Funktionen sind teilweise notwendig, um parallele Datenstrukturen ohne zusätzliche Hardware-Unterstützung zu implementieren. Zum Beispiel erfordert die Implementierung des Software-basierten Locks nach Dekkers Algorithmus eine genaue Ordnung der Lese- und Schreibzugriffe, damit beim Test der jeweils anderen Variable der tatsächliche Zustand der Variable gelesen werden kann. Würde das Schreiben des Variablenwertes verzögert, könnten eventuell beide Threads den kritischen Bereich betreten. Mit Hilfe der hier dargestellten Funktionen kann das für eine korrekte Ausführung notwendige Verhalten garantiert werden.

5.3.2 Parallele Speicherverwaltung „memlist“

Das Framework bietet mit dem Speicherverwaltungsinterface „memlist“ (siehe Kapitel 2.5) die Möglichkeit, Speicherblöcke fester Größe effizient bereitzustellen. Werden innerhalb eines parallelen Programms besonders häufig dynamische Datenstrukturen erzeugt und frei-

```
X86:
  static inline void hw_mem_barrier()
  {
    asm volatile( "mfence" : : : "memory" );
  }

Power:
  static inline void hw_mem_barrier()
  {
    asm volatile( "sync" : : : "memory" );
  }

IA-64 (GCC):
  static inline void hw_mem_barrier()
  {
    asm volatile( "mf;;" : : : "memory" );
  }

IA-64 (ICC):
  static inline void hw_mem_barrier()
  {
    ia64_mf();
  }
```

Abbildung 5.18: Implementierung der Mem-Barrier-Unterstützung für verschiedene Architekturen.

gegeben, kann der Overhead durch die Benutzung von *malloc* und *free* erheblich sein. Daher können beliebig viele Speicherlisten mit *memlist_init* erzeugt werden, die jeweils für Speicherblöcke einer gegebenen Größe verantwortlich sind.

Abbildung 5.19 zeigt die dafür verwendete Datenstruktur. Darin wird die Größe der zu verwaltenden Speicherblöcke und die Anzahl von zugreifenden Threads abgespeichert, wie es bei der Initialisierung angegeben wurde. Zusätzlich wird ein Funktionszeiger gespeichert, der optional die Initialisierung der einzelnen Speicherblöcke übernimmt.

Die Implementierung ist analog zu den in Kapitel 5.1.4 beschriebenen Mechanismen für die Taskpool-spezifischen Operationen. Die Größe der einzelnen Speicherblöcke wird derart angepasst, dass mit einer gegebenen Cache-Zeilengröße kein False-Sharing zwischen Elementen auftritt. Für jeden Thread gibt es in dem Datenfeld „mem“ der „memlist_t“-Struktur einen Eintrag, der jeweils einen großen Verwaltungsblock als Puffer für zu entnehmende Blöcke speichert (Datenfeld „buf“). Ein entsprechender Index gibt die freie Position im Puffer an (Datenfeld „index“). Um häufige Allokationen zu vermeiden, werden freigegebene Speicherelemente in einer Freiliste abgespeichert (Datenfeld „free_items“), um bei Bedarf daraus Elemente zu entnehmen, bevor auf den Puffer zurückgegriffen wird. Das Pufferelement speichert zusätzlich einen Zeiger auf ein eventuelles nächstes Element, sodass neue Puffer allokiert und eingetragen werden können, wenn ein Puffer aufgebraucht wurde.

Die Strukturen für die einzelnen Threads sind im Datenfeld „mem“ derart verteilt, dass auch beim Zugriff darauf kein False-Sharing auftritt. Somit können die Threads ohne Synchronisation und Nebeneffekte Speicherblöcke allokieren und freigeben. Da die Größe der Speicherblöcke so berechnet wurde, dass False-Sharing nicht auftritt, können auch problemlos allokierte Blöcke anderer Threads freigeben und entsprechend in der Freiliste eingetragen

```
typedef struct _memlist_item_t
{
    struct _memlist_item_t *next;
} memlist_item_t;

typedef void (*memlist_item_init_f)( int, void *);

typedef struct
{
    struct _memlist_thread_t {
        memlist_item_t *buf;
        unsigned long index;

        memlist_item_t *free_items;
    } *mem;

    int item_size;
    int p;
    memlist_item_init_f init_function;
} memlist_t;
```

Abbildung 5.19: Datenstruktur der Speicherliste „memlist“.

werden. Dies tritt häufig bei der parallelen Abarbeitung mit Lastbalancierung auf, da ein Thread Berechnungen eines anderen Threads übernehmen kann.

5.3.3 Zeitmessung mit Hardware-Unterstützung „hw_timesnap“

Insbesondere im Bereich der Performance-Analyse und -Vorhersage sind Zeitmessungen eine wichtige Grundlage für den Vergleich verschiedener Algorithmen. Allerdings ist mit der Zeitmessung immer ein Overhead verbunden, der je nach der erforderlichen Granularität der Zeitmessung zu hoch sein kann.

Ein allgemeiner Ansatz zur Zeitmessung von Programmlaufzeiten ist die Benutzung der Funktion „gettimeofday“ für die Bestimmung der aktuellen (Real-)Zeit und „times“ für die Bestimmung der CPU-Zeit (und der System-Zeit). Beide sind in den POSIX-Spezifikationen enthalten und somit portabel. Allerdings sind sie auch mit einem relativ hohen Aufwand verbunden, da die exakte Zeit des Systems bestimmt werden muss. Zudem haben diese Funktionen nur eine relativ geringe zeitliche Auflösung. Die Funktion „gettimeofday“ hat durch die benutzte Datenstruktur eine maximale Auflösung in Mikrosekunden. Die Funktion „times“ gibt Vielfache von „clock ticks“ zurück, ein spezieller Wert für die Auflösung des entsprechenden Zeitgebers. Der Wert schwankt je nach System, häufig beträgt die Auflösung aber nur $10\,ms$ oder $1\,ms$.

Daher kann es sinnvoll sein, auf andere Zeitmessverfahren zurückzugreifen, die weniger Overhead besitzen, genauer messen können, aber auch nicht so portabel sind.

Zugriff auf Zeitregister in modernen Prozessoren

Aktuelle Prozessoren bieten häufig Zugriff auf einen internen Taktzähler, der automatisch vom Prozessor selbst zyklisch erhöht wird, oftmals mit jedem Prozessortakt. Das Auslesen dieser Information ist somit mit relativ wenig Aufwand möglich und die Genauigkeit

```
static inline uint64_t hw_timesnap()
{
    uint64_t val;

    asm volatile( "rdtsc" :
                  "=A" ( val ) );
    return val;
}
```

Abbildung 5.20: Zugriff auf das Zeitregister in der x86-Architektur (64-Bit-Variante).

```
static inline uint64_t hw_timesnap()
{
    uint64_t t;

    asm volatile( "mftb %0"
                  : "=r" ( t ) );

    return t;
}
```

Abbildung 5.21: Zugriff auf das Zeitregister in der Power-Architektur (64-Bit-Variante).

ist durch den Takt gegeben. Da der Takt des Zeitzählers nicht direkt an den Prozessortakt gekoppelt sein muss und zudem für verschiedene Entwicklungsstufen der gleichen Prozessorarchitektur unterschiedlich sein kann, bestimmt das KOALA-Framework zur Konfigurationszeit automatisch den Takt des Zeitgebers.

x86. Die Intel-x86-Architektur bietet den Befehl „rdtsc“ (*read time stamp counter*), der einen 64-Bit-Zähler ausliest. Der Zähler wird mit jedem Prozessortakt erhöht, allerdings kann das auch nur der externe Takt oder ein Grundtakt sein, insbesondere bei variablen Taktfrequenzen.

Der Befehl ist sowohl in der 32-Bit als auch in der 64-Bit-Umgebung benutzbar. Abbildung 5.20 zeigt die Implementierung in der 64-Bit-Variante, die nur einen Befehl benötigt, während in der 32-Bit-Variante der 64-Bit-Zähler auf zwei Register verteilt ist und entsprechend zusammengesetzt wird.

Power. Die aktuelle Power-Architektur, die in den Power4+- und Power6-Prozessoren von IBM implementiert ist, unterstützt ebenso einen entsprechenden Zähler. Während die frühere PowerPC-Architektur einen Echtzeitzähler enthielt, wurden in der neueren Power-Architektur allerdings einige Ansprüche an diesen Zähler aus Effizienzgründen verringert. So ist nicht mehr garantiert, dass sich der Zähler mit jedem Prozessortakt erhöht und Rückschlüsse auf die aktuelle (Real-)Zeit sind daher auch nicht unbedingt möglich. Das genaue Verhalten bleibt der Implementierung der Architektur überlassen, in der Praxis ist meist das Verhalten anzutreffen, dass mit einem vorgegebenen Takt der Zähler kontinuierlich erhöht wird. Die Auswirkungen solcher Unsicherheiten auf die Zeitmessungen werden in den folgenden Abschnitten genauer betrachtet.

```
static inline uint64_t hw_timesnap()
{
    uint64_t val;

    asm volatile( "mov %0=ar.itc" :
                  "=r" ( val ) );

    return val;
}
```

Abbildung 5.22: Zugriff auf das Zeitregister in der IA-64-Architektur.

Abbildung 5.21 zeigt die Implementierung für das Auslesen des Zählers. In einer 64-Bit-Umgebung wird der Befehl „mftb" benutzt, während in einer 32-Bit-Umgebung eine Kombination aus „mftb" und „mftbu" benutzt werden muss. In dieser Variante müssen gegebenenfalls diese beiden Instruktionen wiederholt werden, wenn ein Überlauf auftrat. Entsprechend ist das Auslesen in der 32-Bit-Variante mit etwas höherem Aufwand verbunden.

IA-64. Innerhalb des Itanium-Prozessors ist der Zeitzähler direkt als Register verfügbar, sodass der aktuelle Wert einfach ohne zusätzlichen Befehl heraus kopiert werden kann (Abbildung 5.22).

Da der *asm*-Befehl bei der Nutzung des Intel-Compilers nicht verfügbar ist, muss auf entsprechende Intrinsics-Befehle des Compilers zurückgegriffen werden.

Obwohl das Kommando nur den Wert eines Registers in ein anderes Register kopiert, dauert der Zugriff dennoch bis zu 40 Takte, da der Prozessor dieses Register speziell behandelt [60].

Betrachtungen zur Genauigkeit

Die genannten Befehle auf den verschiedenen Architekturen erlauben das Auslesen eines Zählers in weit weniger als 100 Prozessortakten, die Genauigkeit kann dabei also wenige Nanosekunden erreichen, eventuell ist sogar eine taktgenaue Messung möglich. Allerdings haben diese Befehle bzw. die eigentlichen Zeitzähler gewisse Nebenwirkungen, die bei der Benutzung bewusst berücksichtigt werden müssen, um die hohe Auflösung auch praktisch nutzen zu können. Im Folgenden werden einige Probleme dieser Zeitmessung erläutert, die so bei der Benutzung von Bibliotheksaufrufen wie „gettimeofday" nicht auftreten oder vom Betriebssystem korrigiert werden.

Dynamische Taktanpassung. Ältere Prozessoren wurden mit einem festen Takt gesteuert, wobei existierende Zeitzähler taktgenau mitliefen. Damit war eine genaue Zeitmessung möglich, die auch direkt in eine Real-Zeit umgerechnet werden konnte. Mittlerweile werden aktuelle Prozessoren aber nicht mehr mit einer festen Taktfrequenz betrieben, sondern können aus Energiespargründen mit einer reduzierten Taktfrequenz betrieben werden. Teilweise wird der reguläre Takt sogar erhöht, um kurzzeitig mehr Leistung zu erzielen, wenn es der thermische Zustand des Prozessors erlaubt. Ist der Zeitzähler an den Prozessortakt

gekoppelt, ist die zu einer Zählwertdifferenz entsprechende Real-Zeit also nicht mehr direkt berechenbar. Anders gesagt können zwei gleiche, gemessene Zeitdifferenzen unterschiedliche Real-Zeiten beschreiben. Ein Vergleich zweier Messwerte ist also nur noch bedingt möglich.

Manche Architekturen lösen das Problem, indem der Zähler vom aktuellen Prozessortakt losgelöst wird und beispielsweise nur noch mit einem festgelegten Grundtakt betrieben wird. Allerdings wird dieses Verhalten teilweise nicht garantiert und hängt von der jeweiligen Implementierung ab. Auf manchen Architekturen laufen die Zähler auch bei einer Taktverringerung weiter mit Höchsttakt, sodass die Zähler entsprechend schneller zählen als die CPU tatsächlich Befehle ausführen kann. Im Verlauf dieses Abschnittes werden experimentelle Ergebnisse für unterschiedliche Plattformen aufgelistet, die jeweils unterschiedliche Strategien benutzen. Dies kann sich jedoch mit neuen Prozessorversionen ändern.

Zeitverschiebungen in SMP-Systemen. In einem Mehrprozessorsystem (und entsprechend auch in einem Multi-Core-System) besitzt jeder Prozessorkern einen eigenen Zähler. Diese Zähler sind untereinander jedoch nicht synchronisiert, sodass die absolut gemessenen Werte nur für einen Kern vergleichbar sind. Zudem können die Taktfrequenzen der Kerne teilweise einzeln an den entsprechenden Lastzustand angepasst werden, sodass auch Zeitdifferenzen nicht unbedingt vergleichbar sind. Wird zudem ein Thread vom Betriebssystem auf einen anderen Prozessorkern verschoben, erhält des Benutzerprogramm beim nächsten Auslesen des Zählregisters einen anderen Wert. Differenzen zweier Messpunkte können somit eventuell falsche Werte ergeben.

Es genügt zudem nicht, einmalig zu Beginn der Messungen für alle Kerne den aktuellen Zählerstand auszulesen, da die Zähler unterschiedlichen zeitlichen Drifts unterliegen. Während die Kerne eines Multi-Core-Prozessors meist noch eine ähnliche Zeitbasis haben, kann ein anderer Prozessor in einem SMP-System eine größere Ungenauigkeit erzeugen.

Teilweise können die Probleme eingeschränkt werden, indem Threads an Prozessorkerne gebunden werden, sodass zumindest Ungenauigkeiten durch Wechsel eines Kerns nicht auftreten. Nicht jedes Betriebssystem unterstützt dies jedoch bzw. erlaubt es Programmen beliebiger Benutzer, die Zuordnung vorzugeben.

Eine weitere Unsicherheit ist ein eventuell automatisches Korrigieren der Zeitzähler durch das Betriebssystem. Oft werden die Echtzeituhren vom Betriebssystem durch Abgleich mit anderen Zeitquellen (z. B. Internet-Zeit) nachgestellt. Das Betriebssystem kann entsprechend auch die Hardware-Zeitzähler in den Prozessorkernen entsprechend korrigieren, um z. B. eine Abbildung auf die Real-Zeit zu erlauben. Dies fügt eine weitere Unsicherheit in die Zeitmessung ein.

Praktische Auswirkungen. Hardware-Taktzähler können ein sinnvoller Ersatz für konventionelle und portable Zeitmessverfahren sein. Sie sind schneller auszulesen und oft genauer. Dennoch muss bei einer Implementierung beachtet werden, dass Vergleichbarkeit mit der Real-Zeit und auch zwischen verschiedenen Zeitpunkten nur bedingt gegeben ist. In den in dieser Arbeit ausgeführten Laufzeitexperimenten sind die Einflüsse durch Taktanpassungen relativ klein, da die Prozessoren aufgrund der Auslastung ohnehin mit der normalen Taktfrequenz betrieben werden und wenn überhaupt nur selten langsamer getaktet werden. Wenn bei der Benutzung berücksichtigt wird, dass einige wenige Messungen den oben

dargestellten Fehlerquellen ausgesetzt sind, kann die Ungenauigkeit durch die erwähnten Einflüsse für die höhere Auflösung bei wesentlich geringeren Kosten in Kauf genommen werden. Werden beispielsweise die Messwerte statistisch mit einer Durchschnittsberechnung ausgewertet, spielen einige wenige Ausreißer keine Rolle.

Experimentelle Ergebnisse auf verschiedenen Plattformen

In diesem Abschnitt werden die Resultate von Laufzeittests zur Bestimmung der Auflösung des Zeitzählers auf den verschiedenen Architekturen dargestellt. Dazu wird zuerst direkt hintereinander zweimal der aktuelle Zählerstand ausgelesen, womit also der Overhead für die Zeitmessung eines leeren Codeblocks bestimmt wird. Danach wird die Anzahl der verstrichenen Zähltakte für eine Pause von einer Sekunde gemessen. Aus diesem Messwert lässt sich die Taktfrequenz des Zeitgebers bestimmen.

Auf allen betrachteten Plattformen zeigte sich, dass Wartezeiten (wie hier durch Aufruf von „sleep“) mitgemessen werden. Dies ist für eine spätere Nutzung der Zeitmessung relevant, da das Betriebssystem dies auch anders realisieren könnte. Bei einer Prozessorsuspendierung könnte der aktuelle Zählerstand gespeichert werden und bei der nächsten Fortsetzung wiederhergestellt werden. Da aber Wartezeiten mitgezählt werden, können diese Zähler als Ersatz der echten Systemzeit genutzt werden, um nicht nur die Dauer von Ausführungszeiten zu messen, sondern auch um Wartezeiten finden zu können.

Im Folgenden werden die Ergebnisse der Laufzeittests für die einzelnen Systeme beschrieben.

x86-Architektur. Der Zeitzähler des „Intel Pentium 4“-Prozessors mit 3,0 GHz Taktfrequenz läuft mit dem Prozessortakt. Die Messung des leeren Rumpfes dauerte 84 Taktzyklen, was 28 Nanosekunden entspricht.

Der zweite untersuchte Prozessor ist ein „AMD Opteron Prozessor 244“. Der Prozessortakt ist 1,8 GHz, der Zeitzähler läuft ebenfalls mit dieser Frequenz. Der leere Rumpf dauert hier nur 8 Zyklen bzw. $\approx 4,4$ Nanosekunden.

Auf einem System mit einem „Intel Core2 Duo“-Prozessor mit 3,0 GHz wird die Taktfrequenz in Abhängigkeit von der Last auf bis zu 2,0 GHz reduziert. Der Zähler läuft dennoch immer mit 3,0 GHz. Entsprechend dauert die Messung des leeren Rumpfes bei 2,0 GHz Prozessortakt 54 Zähltakte (18 Nanosekunden) und bei 3,0 GHz 36 Zähltakte (12 Nanosekunden).

Power-Architektur. Die „Power4+“-Prozessoren des „JUMP“-Systems des Forschungszentrums Jülich sind mit 1,7 GHz getaktet. Die Messung ergab eine Taktfrequenz des Zeitzählers von rund 213 MHz, in etwa $\frac{1}{8}$ des Prozessortakts. Die Messung des leeren Rumpfes dauert 3 Zyklen, also rund 14 Nanosekunden.

IA-64-Architektur. Für diese Architektur wurden zwei verschiedene Generationen von Itanium2-Prozessoren untersucht. Die „Itanium2“-Prozessoren vom Modell „Madison“ laufen mit einer Taktfrequenz von 1,5 GHz. Der Zeitzähler wird auf diesem System auch mit 1,5 GHz getaktet. Für den leeren Rumpf wurden 6 Zyklen benötigt, das entspricht 4 Nanosekunden.

Das zweite System im Leibniz-Rechenzentrum München hat „Itanium2“-Prozessoren vom Modell „Montecito“ mit einer Taktfrequenz von 1,6 GHz. Der Zeitzähler ist nur mit 400 MHz getaktet. Der leere Rumpf dauert einen Takt, also 2,5 Nanosekunden.

Verfügbarkeit im Framework

Das *configure*-Skript erkennt automatisch die Architektur und aktiviert die korrekte Implementierung der Hardware-Zeitmessung. Wenn eine Implementierung für die Zielarchitektur jedoch nicht verfügbar ist, kann eine Software-Lösung aktiviert werden. Damit können Komponenten genutzt werden, die eigentlich eine Zeitmessung mit geringem Overhead benötigen. Die Option *--enable-hw-time-sw-fallback* führt zu einer Nutzung von *gettimeofday*, wodurch allerdings sowohl die Auflösung sinkt als auch der Overhead steigt.

Wenn durch die Wahl der einzelnen Komponenten eine Hardware-Zeitmessung notwendig ist, wird das *configure*-Skript versuchen, die Taktrate des Zeitzählers zu bestimmen. Schlägt dies fehl, kann mittels der Option *--disable-itc-test* trotzdem fortgefahren werden. Die mit den Hardware-Zeitmessungen erzeugten Daten haben dann allerdings keine festgelegte Zeitbasis und können entsprechend nicht unbedingt mit anderen Resultaten verglichen werden.

5.4 Automatisches Tuning Hardware-abhängiger Parameter

Spezielle Optimierungen in einigen Komponenten wie z. B. dem Speichermanager versuchen, Cache-Fehlzugriffe durch False-Sharing zu verringern, indem Datenstrukturen geeignet ausgerichtet werden. In Kapitel 5.1.4 wurden schon die beiden Konstanten *TP_COARSE_PADDING* und *TP_FINE_PADDING* beschrieben, die die verwendeten Blockgrößen festlegen. Insbesondere der Wert von *TP_FINE_PADDING* hat starken Einfluss auf Performance und Speicherverbrauch, da die Speicherblöcke für interne Datenstrukturen und applikationsspezifische Datenstrukturen davon beeinflusst werden.

Die Wahl dieses Parameters ist nicht ganz einfach. Ein wesentlicher Aspekt ist die Granularität des Speichersystems. Schreibende Zugriffe auf Adressen innerhalb einer Cache-Zeile durch mehrere Prozessoren führen unweigerlich zu einer Invalidierung in den Caches der jeweils anderen Prozessoren. Bei häufigem Zugriff kann es zu einer erheblichen Anzahl von Fehlzugriffen mit entsprechend größeren Speicherzugriffszeiten kommen. Oft unterscheidet sich jedoch die Zeilengröße je nach Speicherhierarchie und außerdem werden Techniken wie Prefetching von der Hardware automatisch eingesetzt, sodass es auch mit Kenntnis der exakten Parameter der Hardware schwierig ist, einen optimalen Wert für *TP_FINE_PADDING* zu finden. Zudem bedeutet ein größerer Wert zwar immer weniger mögliche Fehlzugriffe, aber eben auch einen wesentlich höheren Speicherverbrauch.

Einerseits möchte man die Wahl nicht dem Nutzer des KOALA-Frameworks überlassen müssen, der eventuell diese Information nicht besitzt oder bestimmen möchte. Andererseits kann der Programmierer einer taskbasierten Applikation nicht unbedingt die Parameter der Zielplattform kennen. Daher stellt das Framework ein automatisches *Tuning* dieses Parameters bereit.

Der Benutzer wird dabei durch die Tuning-Prozedur geleitet. Führt der Benutzer direkt nach der Framework-Konfiguration mit dem „configure“-Skript die ausgewählte Applikation

aus, wird er auf die nötige Tuning-Prozedur hingewiesen:

```
Taskpool Framework not calibrated!
Start with argument "--init-p <threads>" (<threads> >= 2)
```

Da False-Sharing erst bei Benutzung mehrerer Threads aufgedeckt werden kann, müssen wenigstens zwei Threads für die Kalibrierung verwendet werden. Mehr Threads verbessern das Ergebnis, allerdings sollten nicht mehr Threads eingesetzt werden als Prozessoren (bzw. Prozessorkerne) zur Verfügung stehen.

Wurde die entsprechende Anzahl zu benutzender Threads vorgegeben, startet ein automatischer Kalibrierungslauf, der die synthetische Testapplikation mit dem gewählten Taskpool ausführt. Dabei werden leere Tasks ausgeführt, die nur neue Tasks entsprechend dem Schema der Testapplikation erzeugen (der Parameter f ist also 0). Damit wird eine maximale Zugriffsrate auf die Taskpool-Komponenten realisiert.

Im Laufe der Kalibrierung werden verschiedene Ausrichtungsgrößen jeweils mit nur einem Thread und nochmals mit der gewählten Anzahl von Threads getestet. Cache-Fehlzugriffe durch höhere Invalidierungen von Cache-Zeilen führen zu einer höheren User-Zeit des Prozesses durch höhere Speicherzugriffszeiten. Entsprechend wird der Unterschied dieser Messgröße für einen und mehrere Threads als Basis der Entscheidung genutzt, um einen möglichst geringen Anstieg der User-Zeit gegenüber der sequenziellen Ausführung zu erzielen.

Abbildung 5.23 zeigt exemplarisch die (aus Platzgründen gekürzte) Ausgabe des Kalibrierungslaufs mit vier Threads auf einer Itanium2-Architektur. Die langsamere Ausführung einer Testschleife mit mehreren Threads gegenüber der Ausführung mit nur einem Thread dient als Merkmal zur Ermittlung des optimalen Wertes. Im Beispiel benötigt die parallel ausgeführte Taskabarbeitung fast doppelt so lange wie die sequenzielle Abarbeitung, wenn eine Ausrichtungsgröße von 1 gewählt wird, womit effektiv die angeforderte Blockgröße unverändert bleibt. Mit steigendem Wert von *TP_FINE_PADDING* verringert sich die Verlangsamung, wobei im Beispiel erst bei einer Ausrichtungsgröße von 32 Bytes eine merkliche Verringerung der Verlangsamung beobachtet werden kann (33% statt 87% langsamer).

Auf Basis dieser Messungen, die jeweils mehrfach wiederholt werden, wird versucht, eine Empfehlung für den Nutzer zu bestimmen. Der erste Wert gibt die Größe der Ausrichtung an, mit der die beste durchschnittlichen Laufzeit erreicht wurde. Damit nicht mehr Speicher als nötig verwendet wird, wird dazu jedoch nicht der absolut beste Werte gewählt, sondern der Wert für den es keinen besseren Wert gibt, der mindestens eine 2,5 Prozent höhere Leistung erreicht. Dieser Prozentwert wurde empirisch als geeignet bestimmt. Da im Test jeweils der Ausrichtungswert verdoppelt wird, wird eventuell auch entsprechend doppelt so viel Speicher verbraucht. Dies sollte durch eine Mindestverbesserung gerechtfertigt sein.

Der zweite Wert gibt an, welches der beste Wert basierend auf dem Minimum der jeweiligen Messungen ist (wieder innerhalb des 2,5 Prozent-Bereiches). Als Drittes wird der beste Werte (innerhalb des 2,5 Prozent-Bereiches) angezeigt, der in Relation zum Abstand der parallelen zur sequenziellen Ausführung am geringsten ist.

Obwohl diese drei Werte die besten Ergebnisse repräsentieren, genügt es oftmals aber, eine etwas langsamere Ausführung zu akzeptieren, wenn der Speicherbedarf dadurch wesentlich verringert werden kann. Daher wird als zusätzliche Entscheidungshilfe eine Tabelle

```
Calibrating pad size using empty tasks...
Testing pad size 1...
  AVG: Results:2.176000(1) 4.077600(4)
  AVG: Parallel slowdown:87.389706%
  MIN: Results:2.172000(1) 3.852000(4)
  MIN: Parallel slowdown:77.348066%
Testing pad size 2...
  AVG: Parallel slowdown:86.617647%
  MIN: Parallel slowdown:79.227941%
Testing pad size 4...
Testing pad size 8...
Testing pad size 16...
Testing pad size 32...
  AVG: Parallel slowdown:33.786765%
  MIN: Parallel slowdown:30.018416%
Testing pad size 64...
  AVG: Parallel slowdown:15.147059%
  MIN: Parallel slowdown:6.066176%
Testing pad size 128...
  AVG: Parallel slowdown:7.776963%
  MIN: Parallel slowdown:3.119266%
Testing pad size 256...
  AVG: Parallel slowdown:-2.348624%
  MIN: Parallel slowdown:-2.752294%
Testing pad size 512...
Testing pad size 1024...

Found best fine_pad on average:256
Found best fine_pad on minimum:256
Found best fine_pad on average (REL):256

Average results:
                                             5%   10%   20%   50%
  Smallest pad with x% between
  best and worst par. performance      :   256   256    64    32
  Smallest pad with x% between
  sequential and worst performance     :   256   128    64    32
  Smallest pad with x% slowdown        :   256   128    64    32
Minimum results:
                                             5%   10%   20%   50%
  Smallest pad with x% between
  best and worst par. performance      :   256    64    64    32
  Smallest pad with x% between
  sequential and worst performance     :   128    64    64    32
  Smallest pad with x% slowdown        :   128    64    64    32

Set TUNER_USE_FINEPADDING_SIZE in include/tunerconf.h to one
of these values. Set it to 1 to disable aligned data structures
if memory usage matters.
```

Abbildung 5.23: Auszug eines Kalibrierungslaufes auf einer Itanium2-Architektur.

ausgegeben, auf der vier verschiedene Grenzwerte mit jeweils drei verschiedenen Kenngrößen abgebildet werden. Ein Eintrag in der Tabelle gibt jeweils die kleinste Ausrichtungsgröße an, die mit den Parametern der jeweiligen Spalte und Zeile erreicht wird.

Die erste Zeile gibt den Abstand zwischen der schnellsten und der langsamsten parallelen Ausführung an. Die zweite Zeile berücksichtigt den Abstand der schlechtesten parallelen Ausführung zur sequenziellen Zeit, und die dritte Zeile berücksichtigt nur die jeweilige Verlangsamung. Die erste Spalte gibt jeweils die kleinste Ausrichtungsgröße an, deren Ausführungszeit bis auf 5 Prozent an die beste Ausführungszeit heran reicht. Die Testergebnisse werden dazu derart in ein Verhältnis gesetzt, dass Null Prozent der besten Zeit und 100 Prozent der schlechtesten Ausführungszeit entsprechen. Die anderen Spalten geben entsprechend die Werte für 10%, 20% und 50% des Laufzeitspektrums an. Die erste Tabelle gibt die Werte an, die durchschnittlich erreicht wurden, die zweite Tabelle zeigt die entsprechenden Werte, die bei irgendeinem Testlauf minimal erreicht worden.

Die endgültige Festlegung von *TP_FINE_PADDING* wird dem Benutzer überlassen, weil zwar eine komplett automatische Wahl möglich wäre, aber dadurch möglicherweise erheblich mehr Speicher verschwendet wird als unbedingt nötig.

In dem gezeigten Beispiel ist zu sehen, dass erst ab einer Ausrichtungsgröße von 256 Bytes die parallele Ausführung nicht mehr User-Zeit verbraucht als die sequenzielle Ausführung, teilweise sogar etwas weniger. Der Test zeigt, dass man entsprechend mit 256 Bytes gute Ergebnisse erzielt, aber auch mit nur 128 Bytes die Laufzeit immer noch nur 10 Prozent zwischen Minimal- und Maximallaufzeit liegt. Auf diesem Prozessor ist die Cache-Zeilengröße des Level-1-Caches 64 Bytes, während der Level-2- und Level-3-Caches 128 Bytes Zeilengröße verwendet. Dennoch werden die besten Resultate mit 256 Byte erzielt.

Bei der Auswahl des Parameters zur Ausrichtung der Speicherblöcke ist zu beachten, dass der Kalibrierungstest ein Worst-Case-Szenario simuliert, bei dem leere Tasks verarbeitet werden. Die Performance hängt aber auch von der Taskpool-Implementierung ab. Je nach verwendeter Datenstruktur kann eine andere Ausrichtungsgröße empfohlen werden. Für reale Applikationen, die in den Tasks Berechnungen ausführen, kann der Wert weit weniger Einfluss haben in Abhängigkeit davon, wie interne Datenstrukturen aufgebaut sind. Wird z. B. die parallele Speicherliste „memlist“ verwendet, wird auch dort von einer guten Wahl des Parameters profitiert.

Wenn jedoch der Speicherverbrauch durch einen gewählten Wert zu hoch ist, kann ein entsprechend geringerer Wert verwendet werden. Die ausgegebene Tabelle hilft bei der Entscheidung.

5.5 Taskpool-Backend-Implementierungen

Das Taskpool-Backend ist verantwortlich für die Speicherung der von einer Applikation erzeugten Tasks. Es erhält neue Tasks aus der Applikation vom Taskpool-Frontend und gibt ausführungsbereite Tasks auf Anfrage an das Frontend zur Ausführung zurück.

Im Wesentlichen bestimmt das Backend also die verwendete Datenstruktur und die Art und Synchronisation des Zugriffs darauf. Allerdings stellt diese Komponente auch den wesentlichen Anteil an der Gesamtzeit innerhalb des Taskpools dar, sodass gerade hier effiziente Datenstrukturen und Zugriffsmechanismen wichtig sind.

Im Folgenden werden die vorhandenen Implementierungen kurz dargestellt, die zu Vergleichszwecken verwendet werden. Die neuartige adaptive Taskverwaltung wird in Kapitel 6 separat behandelt.

Die Vergleichsimplementierungen basieren auf den Taskpool-Implementierungen aus [69], verwenden jedoch die entsprechenden Komponenten des KOALA-Frameworks für Synchronisation und Speichermanagement.

5.5.1 Grundlagen

Eine Backend-Komponente muss das in Abschnitt 2.6.1 beschriebene Interface implementieren. Die wesentlichen Funktionen sind dabei *tp_intern_put* und *tp_intern_get*. Der Zugriff auf die Backend-Komponente erfolgt parallel, entsprechend muss die Implementierung eigenständig für geeignete Synchronisation sorgen, wenn dies beim Zugriff auf die internen Datenstrukturen nötig ist. Der zugreifende Thread wird dabei durch das Funktionsargument *ID* eindeutig identifiziert, sodass es möglich ist, unterschiedlichen Threads auch unterschiedliche Datenstrukturen zuzuweisen. Neu erzeugte Tasks können entsprechend dem erzeugenden Thread zugewiesen werden und bevorzugt für diesen Thread gespeichert werden. Die Backend-Komponente kann bei der Entnahme auch Tasks so auswählen, dass sie für den abfragenden Threads besonders gut geeignet sind. Dies kann z. B. der Fall sein, wenn Tasks vorher von diesem Thread erzeugt wurden und so mit höherer Wahrscheinlichkeit auf noch vorhandene Daten im Cache zugreifen.

Die Auswahl der zu benutzenden Backend-Komponente erfolgt durch die Wahl der entsprechenden Option des *configure*-Skripts. Mit

```
./configure --enable-<Name>
```

wird die Implementierung mit dem gewählten Namen aktiviert, eine Liste der verfügbaren Implementierungen kann auch ausgegeben werden.

5.5.2 Zentrale Taskpools

Implementierung „sq1“

Diese Implementierung eines zentralen Taskpools entspricht der Implementierung *SQ-PTH* aus Kapitel 4.3.2. Der Taskpool speichert die Tasks in einer Liste ab, die von allen Threads zugegriffen werden kann. Die Tasks sind in einer LIFO-Ordnung abgelegt, sodass neu erzeugte Tasks vor älteren Tasks ausgeführt werden.

Der Zugriff auf die Liste wird durch einen Lock kontrolliert. Da die Implementierung des Locks von der gewählten Lock-Komponente abhängt, ist diese Implementierung nicht direkt vergleichbar mit der Implementierung in Kapitel 4.3.2. Nur bei Benutzung der Pthreads-Lock-Komponente wird der vergleichbare Programmcode benutzt.

5.5.3 Verteilte Taskpools

Implementierung „dq2“

Diese Taskpool-Implementierung entspricht der Variante *DQ-PTH* aus Kapitel 4.3.3. Für jeden Thread existiert eine separate Liste zur Speicherung der Tasks, die jeweils mit einem

eigenen Lock geschützt ist. Neu erzeugte Tasks werden nur in die dem erzeugenden Thread zugeordneten Liste abgespeichert. Wird vom Taskpool-Frontend ein Task für einen Thread angefragt, wird zuerst aus der entsprechenden Liste des Threads ein Task entnommen. Ist die Liste jedoch leer, wird zyklisch beginnend bei dem Thread mit der nächsthöheren ID versucht, dort einen einzelnen Task zu stehlen. Da die Listen komplett öffentlich sind, muss jeder Zugriff auf die Thread-eigene Liste und die Listen der anderen Threads durch Sperrung des entsprechenden Locks geschützt werden.

Die Vergleichbarkeit mit der ursprünglichen Implementierung aus Kapitel 4.3.3 ist wieder nur dann gegeben, wenn als Lock-Komponente die Pthreads-Implementierung gewählt wird.

Implementierung „dq3“

Diese Implementierung, übernommen aus [69], teilt jedem Thread ein Feld fester Größe zur Speicherung der Tasks zu. Ein Index dient zur Adressierung des ersten Elements, sodass neue Tasks jeweils davor abgespeichert werden können und ebenso das jeweils erste Element entnommen werden kann. Durch das vordefinierte Feld wird keine Allokation während der Ausführung notwendig, es wird also nicht auf die Speichermanagement-Komponente zugegriffen.

Threads können bei dieser Implementierung im Unterschied zu den anderen Implementierungen nicht auf die Felder der anderen Threads zugreifen, sodass kein Task-Stealing möglich ist, aber auch keine Synchronisation ausgeführt werden muss.

Diese Implementierung besitzt also einen sehr geringen Overhead, allerdings eignet sie sich auch nur bedingt zur parallelen Taskausführung, da keinerlei Lastbalancierung ausgeführt wird. Daher wird diese Implementierung nur für eine sequenzielle Vergleichsmessung zur Berechnung der Speedups der anderen parallelen Taskpool-Implementierungen verwendet.

5.5.4 Blockverteilte Taskpools

Implementierung „dq8“

Die Implementierung einer blockbasierten Taskspeicherung wurde von der Implementierung *DQ-BL-PTH* aus Kapitel 4.3.4 übernommen. Die Tasks werden zu Blöcken von je vier Tasks zusammengefasst, wobei die LIFO-Tasklisten für jeden Thread einzeln abgespeichert werden. Jede Liste ist durch einen Lock der ausgewählten Lock-Komponente geschützt, die auch Zugriff auf die öffentlichen Listen anderer Threads erlauben (z. B. für Task-Stealing). Im privaten Bereich werden bis zu zwei Blöcke gepuffert. Sind diese vollständig mit Tasks gefüllt, wird einer der beiden in den öffentlichen Bereich verschoben.

Sind bei der Taskentnahme sowohl im öffentlichen als auch im privaten Bereich des entsprechenden Threads keine Tasks in einem Block enthalten, wird zyklisch beginnend bei dem Thread mit der nächsthöheren ID versucht, einen kompletten Block aus dessen öffentlicher Liste zu stehlen. Bei Erfolg wird dieser Block in den zuvor leeren privaten Teil abgespeichert, dieser Block steht also nicht mehr für weitere Stehloperationen anderer Threads zur Verfügung.

5.6 Frontend-Implementierungen

Das Taskpool-Frontend ist das Verbindungsstück zwischen taskbasierter Applikation und den eigentlichen Taskpool-Implementierungen des Backends. Das Frontend stellt Wrapper-Funktionen bereit, um die Backend-Funktionen zur Speicherung von Tasks nutzen zu können. Teilweise sorgt das Frontend für die nötige Synchronisation. Die wichtigste Aufgabe ist jedoch die Steuerung der Taskabarbeitung. Alle Threads betreten durch Aufruf der Frontend-Funktion *tp_run* die Ausführungsschleife, die erst dann verlassen wird, wenn alle Tasks komplett abgearbeitet wurden.

Das Frontend übernimmt keine Verwaltungsaufgaben. Es eignet sich durch die Trennung von der Taskverwaltung im Backend aber auch für die Analyse der taskbasierten Abarbeitung. Daher kann das Frontend auch als Profiling-Komponente implementiert werden. Dort kann dann die Dauer der Programmausführung gemessen werden, aber auch Performance-Parameter einzelner Tasks können bestimmt werden.

Im Folgenden soll jedoch nur die Implementierung der Frontend-Komponente beschrieben werden, die diese Profiling-Möglichkeiten nicht implementiert und so keinen Overhead darstellt. Im Kapitel 7 wird genauer auf die Profiling-Komponente eingegangen, mit der eine genaue Taskanalyse möglich ist.

Die jeweils zu nutzende Frontend-Implementierung mit Namen *<Name>* wird analog zu den anderen Komponenten als Argument *--enable-prof=<Name>* des *configure*-Skripts angegeben.

5.6.1 Frontend-Implementierung „level0“ ohne Profiling

Diese Implementierung stellt die Basiskomponente zur Taskabarbeitung bereit, die den geringsten Einfluss auf die Performance der Applikation ausübt. Diese Implementierung wird daher für Laufzeittests benutzt und führt nur die für die Ausführung von Tasks notwendigen Operationen ohne zusätzlichen Overhead aus.

Genutzte Datenstrukturen. Die Implementierung nutzt zur Synchronisation der beteiligten Threads einen Pthreads-Mutex-Lock und eine Pthreads-Bedingungsvariable. Wenn ein Thread bei einer Anfrage keine Tasks vom Backend erhalten hat, wird er schlafen gelegt und später durch Signalisierung an die Bedingungsvariable wieder aufgeweckt. Außerdem wird eine Barriere vom Typ *tp_barrier_t* (siehe Kapitel 2.4.3) benutzt, um das Betreten der Ausführungsphase zu kontrollieren. So sollen bei einer phasenbasierten Programmabarbeitung keine Threads schon mit der Ausführung der nächsten Phase beginnen, während noch Threads Tasks der vorherigen Phase abarbeiten.

Für eine einfache, applikationsunabhängige Zeitmessung werden optional noch zwei Zeitstempel vom Typ *timesnap_t* (siehe Kapitel 2.5) zu Beginn und Ende der Abarbeitung einer Phase gespeichert.

Initialisierung mit tp_create. Die Erzeugung der Taskpool-Strukturen erfolgt mit der Funktion *tp_create*. Das Frontend stellt dabei sicher, dass dies nur von einem einzigen Thread einmalig zu Beginn ausgeführt wird. Der Aufruf der Initialisierungsfunktion *tp_intern_create* des Taskpool-Backends erfolgt also geschützt durch den Lock des Frontends.

```
 1      tp_barrier( ... );
 2
 3      if ( tp_verbose && ID == 0 ) {
 4          timesnap( ... );
 5      }
 6
 7      for (;;) {
 8          ret_val = tp_intern_get( ID, &function, &argument );
 9          if ( ret_val == TP_NO_TASK ) {
10              WAIT_OR_EXIT;
11              continue;
12          } else if ( ret_val == TP_GOT_TASK_WITH_NEW_ONES &&
13                      tp_ready > 0 ) {
14              pthread_cond_signal( ... );
15          }
16
17          (*(function))( ID, argument );
18
19          tp_free_arg( ID, argument );
20      }
```

Abbildung 5.24: Pseudo-Code-Darstellung der Taskabarbeitungsschleife.

Beendigung mit tp_finalize. Auch die Beendigung des Taskpool-Frameworks erfolgt synchronisiert durch nur einen einzigen Thread, der dann die Funktion *tp_intern_finalize* der Backend-Implementierung aufruft.

Einfügen neuer Tasks in der Initialisierungsphase mit tp_init. Diese Wrapper-Funktion dient dem Einfügen neu erzeugter Tasks während der Initialisierungsphase der Applikation. Dies können also die Starttasks der Applikation sein, die das zu lösende Problem beschreiben. In dieser Frontend-Implementierung wird der Aufruf direkt an das Backend unverändert weitergegeben.

Einfügen neuer Tasks während der Abarbeitung mit tp_put. Auch beim Einfügen neuer Tasks während der Programmausführung wird der Aufruf direkt an die Funktion *tp_intern_put* des Backends weitergegeben. Allerdings müssen zur Laufzeit eventuell schlafende Threads aufgeweckt werden, wenn neue Tasks verfügbar sind.

Daher wird der Rückgabewert der Funktion *tp_intern_put* geprüft, der mit dem Wert *TP_NEW_TASK* diese Situation anzeigt. Wenn zu dem Zeitpunkt mindestens ein Thread pausiert ist, wird einer der wartenden Threads mit Hilfe der Bedingungsvariablen aufgeweckt. Somit kann dann der entsprechende Thread erneut nach ausführungsbereiten Tasks suchen.

Taskabarbeitung mit tp_run. Die wichtigste Aufgabe des Frontends ist die Zuteilung der Tasks an die Threads. Abbildung 5.24 zeigt schematisch den prinzipiellen Aufbau. Nachdem mit Hilfe der Barriere alle teilnehmenden Threads synchronisiert wurden, speichert ein ausgewählter Thread den Start-Zeitstempel, wenn das Framework durch das Kommandozeilenargument *--verbose* eine ausführlichere Ausgabe erzeugen soll.

Die eigentliche Abarbeitungsschleife wird von jedem Thread ausgeführt. Jeder Thread wird vom Backend einen Task mittels Aufruf von *tp_intern_get* erfragen. Wenn kein Task verfügbar ist, wird in Zeile 10 entweder auf die Verfügbarkeit neuer Tasks gewartet oder die Abarbeitung beendet, wenn alle anderen Threads ebenfalls warten und somit keine neuen Tasks erzeugt werden können.

In Zeile 14 wird der Spezialfall behandelt, dass die Entnahme eines Tasks zur Verfügbarkeit neuer Tasks führt. Bei den bisher beschriebenen Backend-Implementierungen tritt dieses Ereignis nicht auf, aber bei den adaptiven Taskpools kann dieses Ereignis auftreten, wenn dynamisch bestimmte Bereiche verändert werden (vgl. Kapitel 6.2). Sollten also neue Tasks verfügbar sein, muss mindestens ein wartender Thread aufgeweckt werden.

Anschließend kann der eigentliche Task in Zeile 17 ausgeführt werden. Unmittelbar danach wird das Taskargument durch den Speichermanager freigegeben.

Wenn der letzte Thread mit der Taskabarbeitung fertig ist und keine weiteren Tasks im Taskpool gespeichert sind, wird die Ausführungsschleife verlassen. Dabei wird nochmals eine Barriere benutzt, damit alle Threads gemeinsam die Taskabarbeitung beenden. Ein ausgewählter Thread gibt anschließend optional die vergangene Real-Zeit, die verbrauchte User- und Systemzeit aus. Außerdem wird die CPU-Auslastung anhand der Formel

$$\text{CPU-Auslastung} = \frac{\frac{\text{User-Zeit}}{\text{Real-Zeit}} * 100.0}{\#\text{Threads}}$$

ausgegeben, die ein einfaches Indiz für die Güte der parallelen Abarbeitung darstellt. Ist dieser Wert wesentlich kleiner als 100 Prozent, traten entsprechend viele Wartezeiten entweder innerhalb der Applikation oder innerhalb des Taskpools z. B. beim Warten auf neue Tasks auf. Eine genauere Untersuchung erlaubt eine erweiterte Frontend-Implementierung, die Profiling-Mechanismen realisiert. Diese werden in Kapitel 7 genauer untersucht.

Kapitel 6

Adaptive Lastbalancierung

Die bisher betrachteten Lastbalancierungsverfahren reduzieren zwar den Verwaltungsaufwand bei der Speicherung der ausführungsbereiten Tasks, allerdings funktioniert das nur dann gut, wenn die einzelnen Threads genügend Tasks zur Ausführung vorrätig haben. Das Task-Stealing selbst, wenn also keine Tasks verfügbar sind, sollte von den Verfahren möglichst so durchgeführt werden, dass die Last schnell balanciert wird.

Die Ansätze mit zentralen Datenstrukturen zur Taskspeicherung können nur für sehr grobgranulare Tasks eingesetzt werden. Bei feingranulareren Tasks und einer größeren Anzahl von Threads dominiert der Synchronisationsaufwand irgendwann die eigentliche Berechnungslaufzeit. Die verteilten Taskpools können im Prinzip mit einer beliebigen Anzahl von Threads betrieben werden. Allerdings steigt hier im Falle einer ungleichen Lastverteilung der Aufwand für das Task-Stealing mit der Anzahl der Threads. Die verteilten Taskpools ohne Gruppierung stehlen nur einzelne Tasks. Wenn eine Applikation allerdings keine neuen Tasks erzeugt, wird das Task-Stealing bis zum Programmende durchgeführt. Die blockverteilten Taskpools umgehen dies, indem mehrere Tasks zu Blöcken zusammengefasst werden. Dadurch werden mehrere Tasks mit einer einzelnen Operation verschoben und der Synchronisationsaufwand sinkt. Allerdings verdeckt die blockweise Speicherung mögliche Parallelität und führt so bei Applikationen mit spezieller Taskstruktur zu erheblicher Verlangsamung, wenn beispielsweise zu einigen Zeitpunkten nur wenige Tasks ausführungsbereit sind. Zudem ist die Blockgröße ein Parameter, der sehr stark von der Applikation abhängt und entsprechend schwer im Voraus zu definieren ist.

Die in diesem Kapitel vorgeschlagene Datenstruktur passt sich dagegen adaptiv an die Gegebenheiten der Applikation an. Sie soll möglichst viele Szenarien taskbasierter Programmabarbeitung gut verwalten. Die gespeicherten Tasks sollen nach Möglichkeit in Blöcken verwaltet werden, um das Task-Stealing effizient realisieren zu können. Dabei sollte die Blockbildung nur dann eingesetzt werden, wenn genügend Parallelität in Form ausführungsbereiter Tasks vorhanden ist. Ein Entwicklungsziel der Datenstruktur ist also die Realisierung dynamischer Blockgrößen, sodass je nach Anzahl der verfügbaren Tasks die Blöcke dynamisch größer oder kleiner werden können.

Adaptive Methoden wurden schon oft verwendet, um Irregularitäten handhaben zu können. In [8] wurde die Effektivität von dynamischem Loop-Scheduling mit adaptiven Techniken untersucht, die allerdings nur auf Schleifen anwendbar sind. [3] schlägt einen adaptiven Scheduler vor, der Laufzeitinformationen benutzt, um dynamisch die Anzahl von Prozessoren zur Ausführung anzupassen. Zur adaptiven Anpassung der gestohlenen Arbeitseinheiten bei einer Lastbalancierung wurde in [51] eine Methode vorgestellt, mit der etwa die Hälfte der Tasks einer Liste entnommen werden kann. Allerdings sind dazu Balancierungsschritte

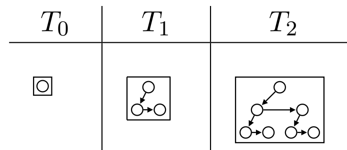


Abbildung 6.1: Schematische Darstellung der Baumstruktur der Tiefe 0, 1 und 2.

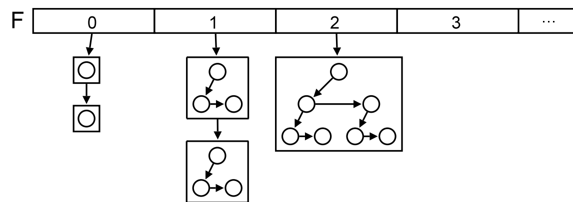


Abbildung 6.2: Beispiel eines teilgefüllten Waldvektors.

nötig, die in regelmäßigen Abständen ausgeführt werden und somit einen entsprechenden Overhead verursachen. Ebenfalls einen adaptiven Ansatz verfolgt die Arbeit in [113], die Berechnungen auf irregulären adaptiven Grids betrachtet. Allerdings sind dort auch periodische Repartitionierungsschritte nötig, um die Last gleichmäßig zu verteilen. Die in dieser Arbeit vorgestellte Struktur benötigt diesen zusätzlichen Aufwand jedoch nicht.

6.1 Adaptive Datenstruktur

Insbesondere bei der Nutzung feingranularer Tasks ist eine wichtige Forderung an die genutzte Datenstruktur schnelle Einfüge- und Entnahmeoperationen. Außerdem sollte die Entnahme einer großen Anzahl von Tasks mit einer einzelnen Operation möglich sein, um den Overhead für die Task-Stealing-Operation zu reduzieren. Eine einfache verkettete Liste benötigt eine konstante Zeit sowohl für das Einfügen neuer Tasks als auch die Entnahme vorhandener Tasks. Allerdings können mehrere Tasks nur mit entsprechend vielen einzelnen Entnahmeoperationen entfernt werden. Bei der Benutzung von Blöcken fester Größe bleibt die konstante Zugriffszeit erhalten und es ist möglich, mehrere Tasks mit einer Operation zu entfernen, effektiv wird damit allerdings nur die Granularität der Tasks erhöht.

Die im Folgenden beschriebene Datenstruktur benutzt dagegen adaptive Blöcke von Tasks, die mit der Anzahl verfügbarer Tasks wachsen oder schrumpfen. Die ausführungsbereiten Tasks werden in voll-balancierten Bäumen gespeichert. Für einen solchen Baum T_i ist also die Länge jedes Pfades von der Wurzel zu einem Blatt genau i. Jedes Blatt und jeder innerer Knoten speichert genau einen Task. Abbildung 6.1 zeigt die Bäume der Tiefe 0, 1 und 2. Ein Knoten des Baums speichert den Zeiger auf den ersten Kindknoten und einen Zeiger zum Nachbarn auf der gleichen Baumebene, sodass der Grad des Baums nicht durch einen festen Wert vorgegeben werden muss.

Die Bäume werden wiederum in einem Waldvektor $F[0..w]$ abgespeichert, der somit einen Wald von vollbalancierten Bäumen darstellt, wobei w die Tiefe des größten zu speichernden Baums ist. Jeder Eintrag $F[i]$ ist ein Zeiger auf eine Liste von Bäumen T_i der Tiefe i, d. h. $F[0]$ speichert nur Bäume mit einer Ebene, also nur einen einzigen Task, $F[1]$ entsprechend

```
Erzeuge neuen Baum t = new T_0 mit:
      t → task = new task
      t → child = NULL
      t → sibling = NULL

if ( Anzahl Bäume in F_0 < l ) {
      t → sibling = F_0
      F_0 = t
} else {
      Suche kleinstes i mit Anzahl Bäume in F_i < l
      t → child = F_{i−1}
      F_{i−1} = NULL
      t → sibling = F_i
      F_i = t
}
```

Abbildung 6.3: Einfügeoperation eines neuen Tasks in den Waldvektor.

nur Bäume der Tiefe 1 mit 3 Tasks usw. (siehe Abbildung 6.2). Da die einzelnen Bäume voll-balanciert abgespeichert werden sollen, können neue Tasks nur durch das Einfügen in $F[0]$ oder durch Kombinieren existierender Bäume gleicher Tiefe zu einem neuen Baum größerer Tiefe eingefügt werden.

Der Grad der Bäume ist nicht festgesetzt und jeder Baumknoten kann prinzipiell einen unterschiedlichen Ausgangsgrad haben. Die Nutzung spezieller Grenzen kann allerdings unterschiedliche Auswirkungen haben. Ein höherer Ausgangsgrad verringert das Wachstum der Bäume, die somit bei gleicher Tiefe mehr Tasks speichern können, aber es existieren entsprechend weniger Bäume. Es ist auch möglich, den Grad der Bäume dynamisch anzupassen, um z. B. den Rechenaufwand einzelner Tasks in die Baumgröße einfließen zu lassen. Für die folgenden Betrachtungen werden der Einfachheit halber Bäume mit einem festen Grad l betrachtet, wobei die Implementierung mit einem Wert $l = 2$ arbeitet.

6.1.1 Einfügen neuer Tasks

Neue Tasks werden entsprechend dem in Abbildung 6.3 dargestellten Algorithmus in den Waldvektor eingefügt. Dazu wird zuerst ein neuer Baum der Tiefe 0 erzeugt, in dessen einzigen Knoten der neue Task gespeichert wird. Wenn in dem Eintrag $F[0]$ des Waldvektors weniger Bäume der Tiefe 0 gespeichert sind, als das Limit l begrenzt, kann der neue Baum direkt dort eingetragen werden. Diese Operation ist also im Wesentlichen eine einfache Einfügeoperation in eine einfach verkettete Liste und entsprechend schnell.

Ist das Größenlimit jedoch erreicht, kann der neue Task nur durch Kombination vorhandener Bäume gespeichert werden. Dazu wird der erste Waldvektoreintrag i gesucht, in dem noch entsprechend dem Limit l Platz zur Speicherung eines weiteren Baums ist. Um einen Baum der nötigen Tiefe i zu erzeugen, wird der neue Task Wurzelknoten aller Bäume aus $F[i-1]$, die nach Definition eine Tiefe von $i-1$ besitzen. Da die Liste der Bäume in $F[i-1]$ schon einfach verkettet ist, kann der erste Eintrag direkt als Kindknoten des neu erzeugten Baums eingetragen werden, der somit die Tiefe i erreicht.

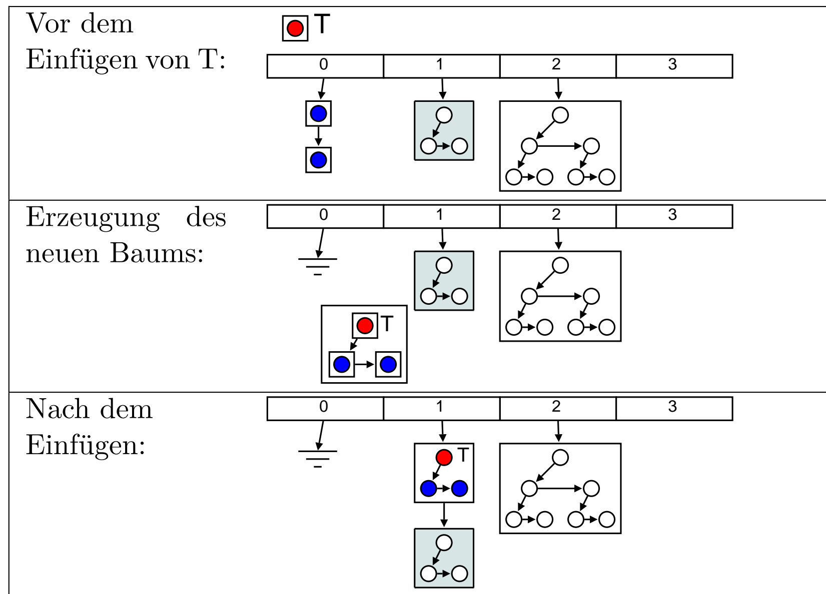


Abbildung 6.4: Beispiel für das Einfügen eines neuen Tasks T.

Suche kleinstes i mit Anzahl Bäume in $F_i > 0$
$t = F_i$
$F_i = t \rightarrow sibling$
if ($i > 0$) {
    $F_{i-1} = t \rightarrow child$
}
return $t \rightarrow task$

Abbildung 6.5: Entnahmeoperation eines Tasks aus einem Waldvektor.

Der neue Baum kann nun in $F[i]$ eingefügt werden, während $F[i-1]$ danach komplett leer ist. Während die Suche maximal w Operationen (Länge des Waldvektors) in Anspruch nimmt, ist die eigentliche Einfügeoperation wiederum eine Operation mit konstanter Komplexität.

Abbildung 6.4 illustriert das Vorgehen für eine gegebene Situation bei einem Limit von $l = 2$. Da der erste Eintrag von F komplett gefüllt ist, muss hier ein neuer Baum erzeugt werden. Im Beispiel ist Eintrag $F[1]$ nur mit einem Baum gefüllt. Daher wird der neue Task T durch Umhängen der Listenzeiger die Wurzel aller bisher in $F[0]$ gespeicherten Knoten. Dieser neue Baum ist ein vollständig balancierter Baum der Tiefe 1 und kann somit vor dem existierenden Baum in $F[1]$ eingefügt werden. Die Liste $F[0]$ ist damit leer und kann die nächsten zwei Tasks direkt aufnehmen.

6.1.2 Entnahme von Tasks

Die Entnahme vorhandener Tasks geschieht im Wesentlichen in umgekehrter Reihenfolge des Einfügens. Tasks aus $F[0]$ können direkt entnommen werden, andere Bäume müssen

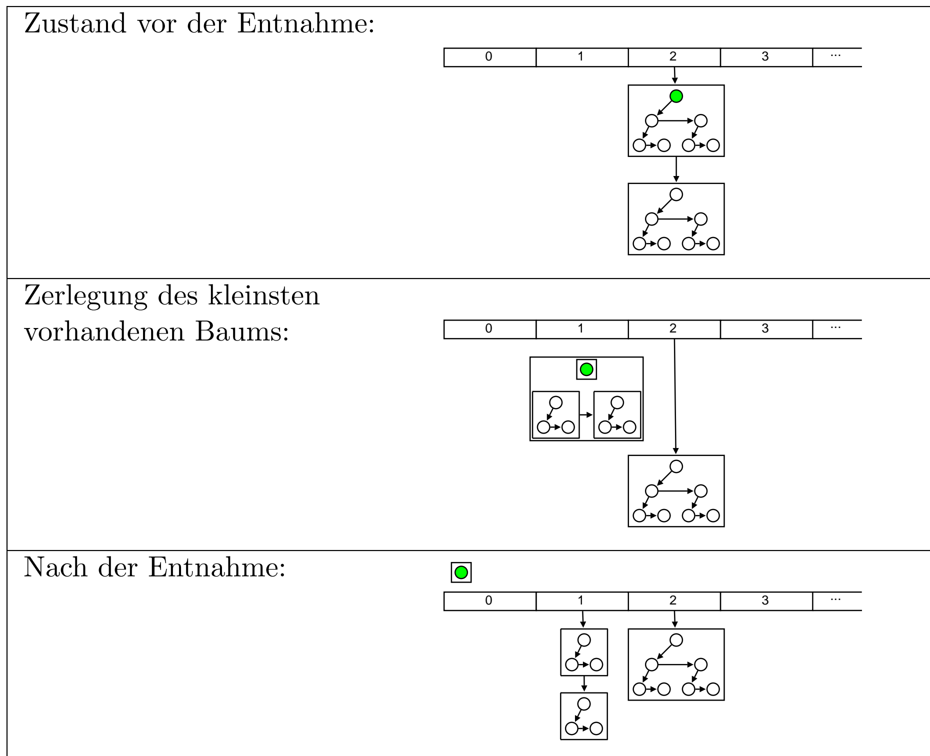


Abbildung 6.6: Beispiel für die Entnahme eines Tasks.

allerdings vor der Entnahme in Teilbäume zerlegt werden. Abbildung 6.5 zeigt den dazu angewendeten Algorithmus, der aus Effizienzgründen keine strikte LIFO-Ordnung sicherstellt und somit keine echte Umkehroperation zum Einfügen ist.

Zur Entnahme wird nach dem ersten Waldvektoreintrag gesucht, der mindestens einen Baum enthält, beginnend beim kleinsten Eintrag $F[0]$. Der Wurzelknoten beschreibt direkt den zu entnehmenden Task. Wird jedoch aus einem Eintrag $F[i]$ mit $i > 0$ ein Task entnommen, so hat der als Task entnommene Wurzelknoten noch eine Menge von Kindknoten, die weiterhin abgespeichert werden müssen. Da $F[i]$ der erste nicht leere Eintrag ist, muss $F[i-1]$ leer sein. Da alle Kindknoten nach Definition Bäume der Tiefe $i-1$ sind und alle Kindknoten miteinander einfach verkettet sind, kann der Zeiger auf den ersten Kindknoten direkt als erstes Listenelement in $F[i-1]$ eingetragen werden. Damit benötigt diese Operation bis auf das Auffinden des Baums konstanten Aufwand. Abbildung 6.6 illustriert die Entnahme eines Tasks für die dargestellte Situation. Der kleinste Baum ist ein Baum der Tiefe 2. Nach der Entnahme dieses Baums wird der Wurzelknoten daraus entnommen und die beiden restlichen Bäume der Tiefe 1 entsprechend in $F[1]$ eingefügt.

Da ein wesentlicher Anteil der Rechenzeit für die Suche nach dem Baum sowohl beim Entnehmen als auch beim Einfügen anfällt, entnimmt die Entnahmeoperation Tasks möglichst weit vorn im Waldvektor, um die Kosten für die Suche zu reduzieren. Wären beispielsweise die ersten zehn Einträge im Waldvektor voll belegt, muss die Einfügeoperation entsprechend weit laufen, um die Bäume des zehnten Eintrags zu einem neuen Baum zu kombinieren.

```
Suche größtes i mit Anzahl Bäume in F_i(p_2) > 0
Entferne Baum von p_2 :
        t = F_i(p_2)
        F_i(p_2) = t → sibling

Füge Baum in F(p_1) ein und zerlege Baum:
        if ( i > 0 ) {
                F_{i−1}(p_1) = t → child
        }

return t → task
```

Abbildung 6.7: Task-Stealing-Operation eines Baums von Thread p_1 zu p_2.

Würde die Entnahmeoperation die genaue Umkehroperation darstellen, müsste auch sie über alle zehn Einträge laufen und den erzeugten Baum zerlegen. Um diesen Aufwand zu reduzieren, wird auf eine LIFO-Ordnung verzichtet. Dies könnte zwar einen negativen Einfluss auf die Datenlokalität haben, da eventuell ältere Tasks vor neueren Tasks ausgeführt werden. Eine garantierte LIFO-Ordnung hätte aber in jedem Fall einen höheren Aufwand bei jeder einzelnen Einfüge- und Entnahmeoperation zur Folge. Auch bei einer großen Anzahl von gespeicherten Tasks, wofür diese Datenstruktur besonders geeignet sein soll, sollen die typischen Operationen der Entnahme und des Einfügens sehr schnell ausgeführt werden. Mit dem vorgeschlagenen Algorithmus arbeiten diese beiden Operationen häufiger am Anfang des Waldvektors, wodurch die Kosten für die Suche innerhalb des Vektors reduziert werden.

6.1.3 Task-Stealing

Zur Lastbalancierung müssen Tasks aus den Datenstrukturen anderer Threads entnommen werden. Prinzipiell könnte dazu die schon beschriebene Entnahmeoperation verwendet werden, wobei allerdings kein Vorteil gegenüber konventionellen Listen bestehen würde. Die adaptive Datenstruktur soll auch die Situation der ungleichen Lastverteilung gut handhaben. Daher wird der in Abbildung 6.7 beschriebene Algorithmus verwendet, um eine leere Datenstruktur wieder mit Tasks zu füllen.

Ziel ist es, möglichst viele Tasks mit einer einzelnen Operation von einer Datenstruktur in eine andere zu verschieben. Daher sucht Thread p_1 in der Datenstruktur eines anderen Threads p_2 nicht am Anfang, sondern am Ende des Waldvektors nach verfügbaren Bäumen. Aus dem nicht leeren Eintrag mit dem größtmöglichen Index i wird entsprechend ein Baum durch Umhängen der Listenzeiger entnommen. Analog zu der normalen Entnahmeoperation wird der gestohlene Baum zerlegt, wobei der Wurzelknoten den nächsten auszuführenden Task darstellt. Ist $i > 0$, werden die restlichen Teilbäume der Tiefe $i - 1$ in $F[i - 1]$ des Threads p_1 abgelegt und sind somit für weitere Entnahmen verfügbar.

Abbildung 6.8 illustriert die Schritte für eine gegebene Situation, bei der Thread 2 keine Tasks mehr gespeichert hat und bei Thread 1 einen Baum sucht. Sei der dort in $F[2]$ gespeicherte Baum der größte, d. h. alle weiteren $F[i], i > 2$ sind leer, so wird dieser aus der

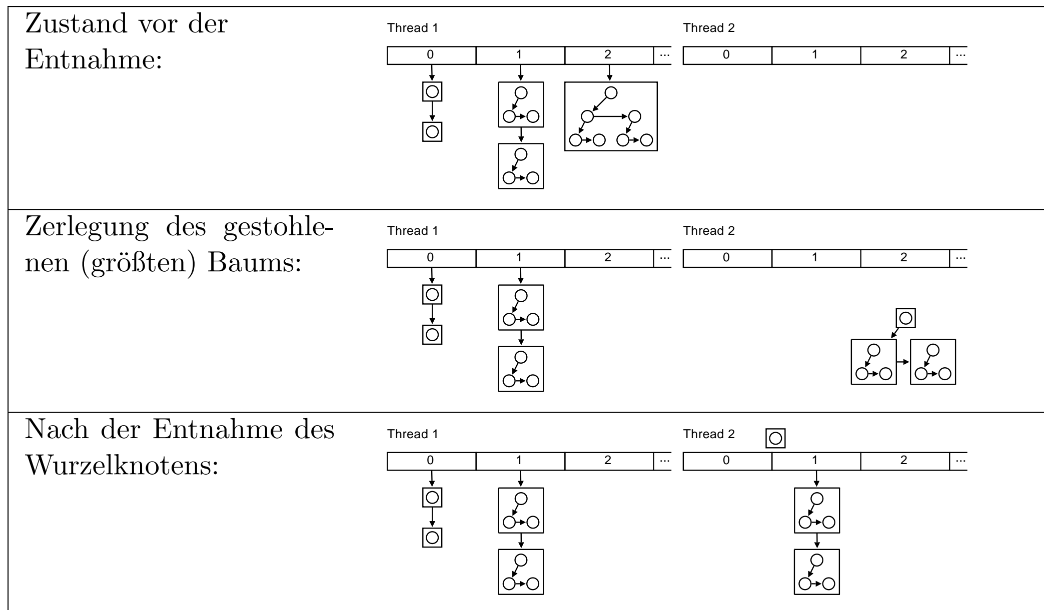


Abbildung 6.8: Beispiel für die Stehleoperation von Thread 2.

Datenstruktur von Thread 1 herausgenommen. Die beiden Teilbäume der Tiefe 1 werden in $F[1]$ der Datenstruktur von Thread 2 gespeichert, während der Wurzelknoten direkt zur Ausführung benutzt wird. Während in diesem Beispiel vor der Operation alle Tasks nur bei Thread 1 gespeichert waren, wurde mit nur einem Schritt die Last nahezu komplett balanciert.

Die Bäume setzen sich jeweils nur aus Bäumen geringer Tiefe zusammen und obwohl die Wurzelknoten der jeweiligen Teilbäume aufgrund der Einfügeoperation wesentlich später erzeugt sein können, so sind doch die Tasks eines Baums der Tiefe i größtenteils älter als Bäume der Tiefe $i - 1$ (in der gleichen Datenstruktur). Ein Thread entnimmt mit dem größten Baum auch Tasks, die relativ weit in der Vergangenheit erzeugt worden sind. Die Wahrscheinlichkeit, dass die verwendeten Daten für die gespeicherten Tasks noch im Cache des bestohlenen Threads liegen, ist also kleiner als für die neueren Tasks in den kleineren Bäumen. Es ist zu erwarten, dass sich der negative Einfluss durch das Task-Stealing bei dieser Vorgehensweise nicht so stark auf die Performance auswirkt.

6.1.4 Komplexitätsbetrachtungen

Offensichtlich hängt die Zahl der mit einer Operation gestohlenen Tasks von der Größe der einzelnen Bäume ab. Für eine genaue Aussage wird im Folgenden die Zahl der gespeicherten Tasks bei der Benutzung eines festen Limits l für den Grad der Bäume abgeschätzt.

Da die einzelnen Bäume vollständig balanciert und gefüllt sind und bei dieser Betrachtung einen festen Ausgangsgrad haben, kann die Anzahl $t(i)$ der in einem Baum der Tiefe i gespeicherten Tasks entsprechend der Formel für geometrische Zahlenfolgen mit

$$t(i) = \frac{l^{i+1} - 1}{l - 1}$$

berechnet werden. Betrachtet man einen bis zur Ebene w vollständig gefüllten Waldvektor $F[0..w]$, so setzt sich die Gesamtzahl der dort gespeicherten Tasks t_{sum} entsprechend der Formel

$$t_{sum} = \sum_{i=0}^{w} l \cdot t(i)$$

zusammen, da in jedem Eintrag i genau l Bäume der Tiefe i gespeichert sind. Somit ist die Gesamtzahl der Tasks

$$\begin{aligned}
t_{sum} &= \sum_{i=0}^{w} l \cdot \left(\frac{l^{i+1} - 1}{l - 1}\right) = \frac{l}{l-1} \sum_{i=0}^{w} \left(l^{i+1} - 1\right) \\
&= \frac{l}{l-1} \left(\sum_{i=0}^{w} l^{i+1} - \sum_{i=0}^{w} 1\right) = \frac{l}{l-1} \left(\sum_{i=0}^{w} l^{i+1} - (w+1)\right) \\
&= \frac{l}{l-1} \left(\sum_{i=0}^{w+1} l^{i} - 1 - (w+1)\right) = \frac{l}{l-1} \left(\frac{l^{w+2} - 1}{l-1} - 1 - (w+1)\right) \\
&= \frac{l}{l-1} \left(\frac{l^{w+2} - 1}{l-1} - \frac{l-1}{l-1} - \frac{(w+1)(l-1)}{l-1}\right) \\
&= \frac{l}{l-1} \left(\frac{l^{w+2} - (w+2)l + (w+1)}{l-1}\right) \\
&= \frac{l^{w+3} - (w+2)l^2 + (w+1)l}{(l-1)^2} \qquad (6.1)
\end{aligned}$$

Der Anteil f der Tasks, die mit einer einzelnen Operation gestohlen werden, lässt sich somit berechnen durch

$$f = \frac{t(w)}{t_{sum}}$$

da immer der jeweils größte Baum entnommen wird. Eingesetzt folgt

$$\begin{aligned}
f &= t(w) \cdot \frac{1}{t_{sum}} \\
f &= \frac{l^{w+1} - 1}{l - 1} \cdot \frac{(l-1)^2}{l^{w+3} - (w+2)l^2 + (w+1)l} \\
&= (l-1) \frac{l^{w+1} - 1}{l^{w+1}} \frac{1}{l^2 - \frac{(w+2)l^2 - (w+1)l}{l^{w+1}}}.
\end{aligned}$$

Der größte Wert für f ist $f = 1/l$ für $w = 0$ und der kleinste Wert für f ist $f = \frac{l-1}{l^2}$ für $w \to \infty$. Bei der Verwendung eines binären Baums ($l = 2$, wie in den Laufzeittests untersucht) werden also mit einer einzelnen Operation zwischen $1/4$ und $1/2$ der gesamten Tasks

in einem Waldvektor gestohlen. Ist der Waldvektor entgegen der Annahme nicht komplett gefüllt, kann der Anteil entsprechend höher sein bis maximal $f = 1$. Die untere Grenze bleibt für beliebig gefüllte Waldvektoren natürlich unverändert gültig. Durch die Zerlegung nach der Stehleoperation können unmittelbar danach ungefähr $1/l$ der gestohlenen Tasks wieder von anderen Threads in Form der l Teilbäume gestohlen werden.

Die obigen Berechnungen betrachten einen einzelnen Waldvektor, verwendet man bei verteilten Taskpools entsprechend mehrere, so ist jeder einzelne unterschiedlich gefüllt und entsprechend unterschiedlich ist das jeweilige w.

Selbst wenn alle Tasks nur bei einem einzigen Thread liegen, können die Tasks in wenigen Operationen verteilt werden, unabhängig von der Menge der Tasks.

Das Einfügen und Entfernen von Tasks kann im „Worst-Case“ $O(\log t_{sum})$ Schritte benötigen, da die Zahl gespeicherter Tasks exponentiell mit der Länge des Waldvektors wächst. Da die Operationen aber so implementiert sind, das immer an der kleinstmöglichen Position im Waldvektor eingefügt und an der kleinstmöglichen Position entnommen wird, sind die Kosten im Mittel wesentlich geringer. Die Task-Stealing-Operation würde entsprechend auch $O(\log t_{sum})$ Schritte benötigen, um den größten Baum zu finden. Speichert man allerdings für jeden Waldvektor das größte $i \leq w$, für das $len\,(F_i) > 0$ gilt, so ist die Operation in konstanter Zeit ausführbar.

6.2 Adaptive Taskpool-Implementierung

Der adaptive Taskpool „atp“ nutzt die oben beschriebene Datenstruktur mit einem festen Limit $l = 2$ und einer Waldvektorgröße von 32 Einträgen, also $w = 31$, wodurch in einer Datenstruktur entsprechend Formel (6.1) über 17 Milliarden Tasks abgespeichert werden können. Jeder Thread erhält eine eigene Instanz dieser Datenstruktur. Der Zugriff auf die Datenstruktur wird durch den Duolock (vgl. Abschnitt 2.4.2) kontrolliert, da zu erwarten ist, dass wesentlich seltener Tasks gestohlen als normal entnommen werden und somit der Datenstruktur-Eigentümer mit möglichst wenig Overhead die Datenstruktur modifizieren kann. Nur die stehlenden Threads müssen den äußeren Lock anfragen, der auf die Lock-Komponente des Taskpool-Frameworks zurückgreift.

Wenn ein Thread keine Tasks mehr zur Ausführung in seiner Datenstruktur hat, sucht er in einer vorgegebenen Reihenfolge in den Datenstrukturen der anderen Threads nach verfügbaren Taskbäumen. Entsprechend der zuvor beschriebenen Operation wird der entnommene Baum in zwei Teilbäume zerlegt und im eigenem Waldvektor abgespeichert. Der im Wurzelknoten gespeicherte Task wird dann direkt ausgeführt.

Generell kann das Stehlen von Tasks anderer Threads negative Auswirkungen auf die Performance haben, da mit höherer Wahrscheinlichkeit benötigte Daten nicht im Cache des stehlenden Threads liegen und entsprechend von entferntem Speicher geladen werden müssen. Durch das Stehlen eines großen Blocks von Tasks kann dieser Effekt reduziert werden, da die in einem Baum abgelegten Tasks möglicherweise auf ähnliche Daten zugreifen. Obwohl die Tasks unabhängig sind, werden sie doch vom erzeugenden Task nicht willkürlich erzeugt, sondern werden benutzt, um Teil- oder Folgeprobleme des aktuell berechneten Problems zu lösen. Entsprechend höher ist die Wahrscheinlichkeit, Daten wiederverwenden zu können. Beispielsweise können nacheinander erzeugte Tasks in einer Ray-Tracing-

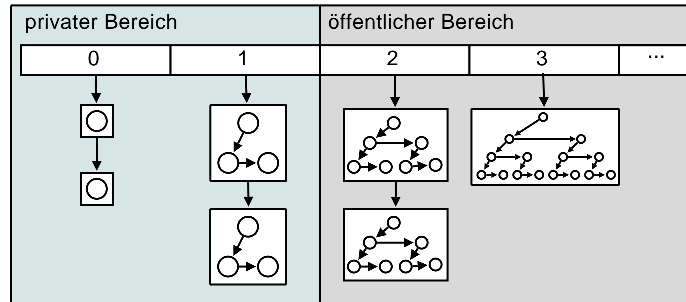


Abbildung 6.9: Beispiel einer Unterteilung des Waldvektors in einen privaten und öffentlichen Bereich.

Applikation benachbarte Pixel berechnen. Dabei werden die Strahlen in ähnliche Bereiche der Szene verfolgt und greifen so eventuell auch auf ähnliche Daten zu. Findet eine Taskerzeugung bei einer Reflexion statt, ist es wiederum wahrscheinlich, dass zwei benachbarte Strahlen auch in einem ähnlichen Bereich der Szene reflektiert werden.

Daher kann ein Baum im Waldvektor potenziell zusammenhängende Tasks enthalten, sodass es zwar zu Cache-Fehlzugriffen kommen kann, aber die Folgetasks wiederum auf dann schon vorhandene Daten zugreifen können. Zwar können durch die Zerteilungs- und Verschmelzungsoperationen verschiedene Teilbäume von verschiedenen Threads erzeugt worden sein, dennoch können einige Teilbäume mit höherer Wahrscheinlichkeit zusammenhängende Tasks enthalten.

Um die Datenlokalität weiter zu verbessern, suchen die Threads jeweils beginnend in ihrer Nachbarschaft nach Tasks, wobei diese anhand ihrer ID definiert ist. Damit soll verhindert werden, dass wenn ein Thread viele Tasks speichert, auch alle anderen Threads nur von ihm Tasks stehlen. Wenn ein anderer Thread von diesem Thread durch eine Operation mindestens 1/4 der Tasks gestohlen hat, werden nun einige Threads wiederum eher von dem stehlenden Thread stehlen, da er näher zu den anderen ist.

Eine zweite adaptive Taskpool-Implementierung „atp2“ unterteilt den Waldvektor in einen privaten und einen öffentlichen Bereich, sodass auf den privaten Teil ohne Synchronisationsoperationen zugegriffen werden kann und entsprechend der Overhead des Zugriffs verringert ist. Die Größe des privaten Bereichs ist ein Kompromiss zwischen dem verringerten Synchronisationsaufwand und der verfügbaren Parallelität. Speichert der private Teil einen großen Anteil der Tasks, muss zwar weniger häufig auf den öffentlichen Teil unter Benutzung der Synchronisationsoperationen zugegriffen werden, aber die Tasks können nicht von anderen Threads gestohlen werden.

Der Waldvektor ermöglicht jedoch eine einfache dynamische Anpassung des privaten und des öffentlichen Bereichs. Mit Hilfe zweier Parameter werden diese Bereiche verwaltet. Der Waldvektor F wird dabei in einen vorderen privaten Bereich $F[0..priv_length-1]$ und einen darauf folgenden öffentlichen Bereich $F[priv_length..w]$ unterteilt (Abbildung 6.9). Der private Bereich liegt also dort, wo bevorzugt Einfüge- und Entnahmeoperationen des Eigentümer-Threads ausgeführt werden, während der öffentliche Bereich die größeren Bäume enthält, die eventuell gestohlen werden können.

Damit möglichst immer Bäume zum Stehlen bereitstehen, sollte der private Bereich nie

den kompletten Waldvektor enthalten. Daher ist der private Bereich durch den Parameter *MAX_PRIVATE_LENGTH* begrenzt, sodass

$$0 \leq priv_length \leq MAX_PRIVATE_LENGTH$$

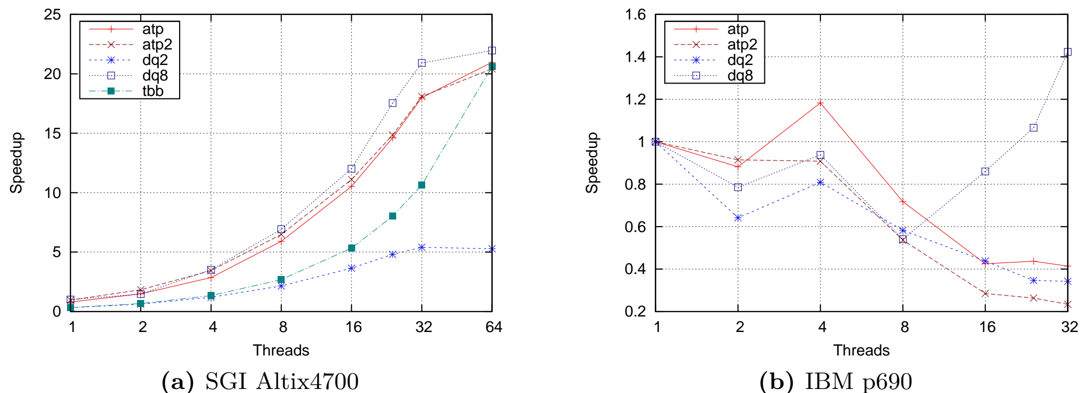
gilt. Ein zweiter, wichtigerer Parameter *MIN_PUBLIC_LENGTH* gibt die minimale Größe des öffentlichen Bereiches vor, sodass der private Bereich erst dann innerhalb der zuvor genannten Grenzen verfügbar ist, wenn

$$pub_length \geq MIN_PUB_LENGTH$$

gilt. In anderen Worten bedeutet dies, dass wenigstens *MIN_PUBLIC_LENGTH* viele Einträge im Waldvektor öffentlich verfügbar sein müssen, bevor höchstens die ersten *MAX_PRIVATE_LENGTH* Einträge als privat markiert werden. Somit soll die Verfügbarkeit der Tasks und damit die Parallelität wichtiger sein als eingesparte Synchronisationsoperationen. Die obere Grenze für den privaten Bereich wurde aus Effizienzgründen gewählt, damit nicht zu häufig diese Grenze aktualisiert werden muss. Wenn die obere Grenze des privaten Bereiches erreicht ist, sind für weitere Einfüge- und Entnahmeoperationen keine Anpassungen nötig, solange die untere Grenze des öffentlichen Bereiches erfüllt ist. Die aktuelle Implementierung verwendet als untere Grenze für den öffentlichen Bereich $MIN_PUBLIC_LENGTH=2$ und als obere Grenze $MAX_PRIVATE_LENGTH=3$ für den privaten Bereich. Sobald also die ersten fünf Einträge des Waldvektors mit mindestens einem Baum belegt sind, können weitere Tasks ohne Veränderung dieser Werte eingefügt werden.

Die Anpassung des privaten Bereichs zur Erfüllung der genannten Grenzen wird nur durch den Datenstruktureigentümer ausgeführt, da ansonsten andere Threads auf den privaten Bereich zugreifen könnten und somit Synchronisation nötig wäre. Während stehlende Threads die Größe des öffentlichen Bereichs entsprechend anpassen, prüft der Eigentümer, ob die untere Grenze für den öffentlichen Bereich unterschritten wurde und passt entsprechend die Größe des privaten Bereichs an. Da er dies jedoch nur zwischen der Taskabarbeitung ausführen kann, kann die Situation eintreten, dass der öffentliche Bereich leer wird, obwohl noch Tasks verfügbar sind. Bei der nächsten Entnahme durch den Eigentümer wird jedoch die Grenze angepasst, sodass Bäume aus dem privaten Bereich wieder verfügbar sind. Deshalb können auch bei der Entnahme von Tasks neue Tasks verfügbar werden, sodass ein Aufwecken anderer Threads notwendig sein kann (siehe Abschnitt 5.6.1).

Die adaptiven Taskpools profitieren also im Gegensatz zu den konventionellen Implementierungen von der Erzeugung einer großen Anzahl von Tasks. Die Lastbalancierung funktioniert in beiden beschriebenen Implementierungen „atp“ und „atp2“ unabhängig von der Anzahl der gespeicherten Tasks in nur wenigen Operationen. Die Implementierung mit einem privaten Bereich erlaubt das synchronisationsfreie Modifizieren der Datenstruktur, wenn genügend Tasks vorhanden sind. Der Overhead zur Speicherung und Ausführung der Tasks sinkt also mit einer steigenden Anzahl gespeicherter Tasks und entsprechend kann die Applikation effizienter arbeiten, wenn sie besonders viele Tasks erzeugt.



(a) SGI Altix4700 **(b)** IBM p690

Abbildung 6.10: Speedups der synthetischen Applikation für leere Tasks ($f = 0$) mit adaptiven Taskpools.

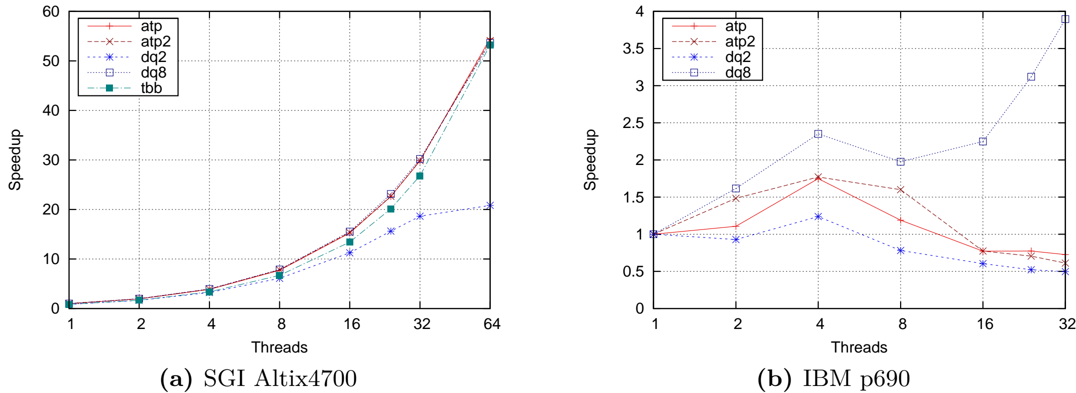
6.3 Experimentelle Ergebnisse

Die Laufzeittests wurden auf einer SGI Altix 4700 Maschine mit 4864 Dual-Core Itanium2-Prozessoren mit einer Taktfrequenz von 1,6 GHz und einer IBM p690 Maschine mit 32 Power4+-Prozessoren mit 1,7 GHz ausgeführt. Als Testanwendungen werden die Applikationen aus Kapitel 3 verwendet. Für die Lock-Komponente wurde die auf allen Systemen verfügbare Pthreads-Locks-Implementierung („mutexlock“) verwendet.

Die adaptiven Taskpool-Implementierungen „atp“ und „atp2“ werden mit dem verteilten Taskpool „dq2“ (*DQ*) und dem blockverteilten Taskpool „dq8“ (*DQ-BL*) verglichen. Für die synthetische Applikation und das parallele Quicksort-Verfahren wird eine vergleichbare Implementierung mit TBB (Threading Building Blocks [111]) untersucht, die allerdings in C++ geschrieben ist und so nur bedingt vergleichbar ist. Zudem ist TBB zum Zeitpunkt der Untersuchungen nicht für die Power-Architektur verfügbar. Für die Berechnung des Speedups wird die beste sequenzielle Laufzeit der betrachteten Implementierungen verwendet.

6.3.1 Synthetische Applikation

In Abbildung 6.10 sind die Ergebnisse für die Benutzung leerer Tasks dargestellt (Parameter $f = 0$), womit also der Overhead der Lastbalancierung gemessen werden kann. Auf der SGI-Maschine (Abbildung 6.10a) werden verhältnismäßig gute Speedups erreicht, allerdings ist der Speedup bei 64 Threads nicht größer als 22. Bei den Messungen erreicht der blockverteilte Taskpool die besten Resultate, da der Overhead für das Taskeinfügen und -entnehmen geringer ist als bei den anderen Implementierungen. Allerdings sind die adaptiven Taskpools schon schneller als der verteilte Taskpool ohne Gruppierung. Die TBB-Implementierung hat zwar wesentliche geringe Speedups, skaliert aber zu sich selbst gemessen besser, was aber aufgrund der insgesamt langsameren Ausführungszeit nicht direkt vergleichbar ist. Bei 64 Threads kann auch der blockverteilte Taskpool keine großen Verbesserungen mehr erreichen. Der Grund dafür liegt in der geringen Anzahl verfügbarer Tasks zu Beginn der Applikation. Durch Vorgabe des Parameters $t = 35$ existieren zu Beginn



(a) SGI Altix4700 (b) IBM p690

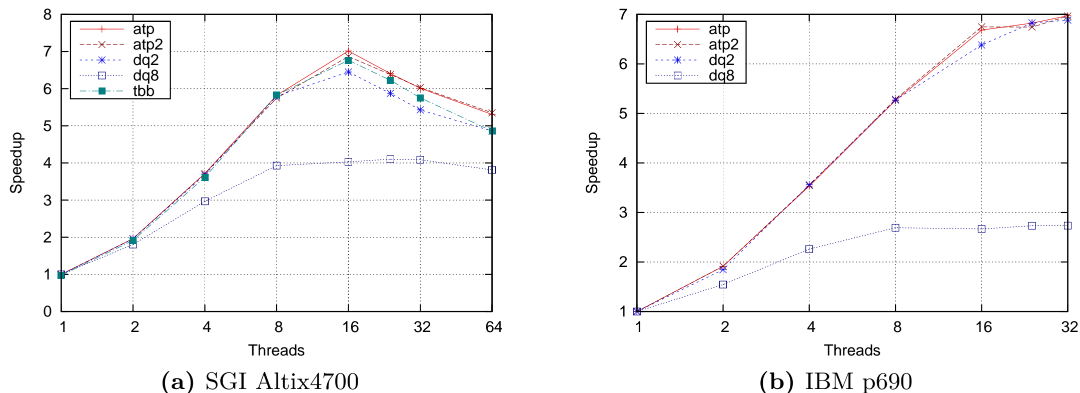
Abbildung 6.11: Speedups der synthetischen Applikation für kleine Tasks ($f = 10$) mit adaptiven Taskpools.

nur 35 Tasks, entsprechend müssen die restlichen Threads warten, bis neue Tasks erzeugt werden. Dies geschieht zwar recht schnell, aber viele der erzeugten Tasks werden in Blöcken des blockverteilten Taskpools gespeichert und sind so noch nicht verfügbar. Bei den adaptiven Taskpools sind die neuen Tasks früher verfügbar und somit fällt weniger Wartezeit an. Der Abstand zum besseren blockverteilten Taskpool verringert sich entsprechend. Bei diesem Test führt die Nutzung von privaten Bereichen bei der adaptiven Implementierung „atp2“ zu keiner signifikanten Laufzeitverbesserung, da der Overhead für die Anpassung der Grenzen hinzukommt.

Auf der IBM-Maschine (Abbildung 6.10b) ist zu sehen, dass die Skalierbarkeit nicht gegeben ist, was aber schon in den Kapiteln 4.3.3 und 4.3.4 beobachtet wurde. Dennoch ist zu erkennen, dass die adaptiven Implementierungen verhältnismäßig gute Ergebnisse erzielen und teilweise sogar schneller sind als der blockverteilte Taskpool. Dieser Taskpool liefert erst ab 16 Threads bessere Ergebnisse, da der steigende Synchronisationsaufwand für mehr Threads aufgrund der Blockbildung weniger stark ins Gewicht fällt. Auch hier ist die Nutzung von privaten Bereichen im adaptiven Taskpool nicht vorteilhaft und sogar etwas langsamer als der verteilte Taskpool ohne Gruppierung.

Wird die Taskgröße etwas erhöht (Abbildung 6.11, $f = 10$), ist die Skalierbarkeit auf der SGI-Maschine (Abbildung 6.11a) schon wesentlich besser und die Taskpools erreichen Speedups von über 50. Nur der verteilte Taskpool ohne Gruppierung kann wie bei leeren Tasks bei mehr als 32 Threads nicht mehr mithalten. Der Abstand der TBB-Implementierung ist aufgrund der größeren Tasks geringer. Die schnellste Implementierung ist jedoch der adaptive Taskpool „atp“.

Auf der IBM-Maschine (Abbildung 6.11b) verbessert sich die Skalierbarkeit etwas, der blockverteilte Taskpool erzielt die verhältnismäßig besten Ergebnisse, während die adaptiven Implementierungen wieder etwas besser sind als der verteilte Taskpool ohne Gruppierung. Dieses Ergebnis überrascht nicht, da in Kapitel 4.3.4 festgestellt wurde, dass auf dieser Maschine selbst Hardware-nahe Taskpool-Implementierungen erst bei einem Parameter $f \approx 50$ gute Ergebnisse liefern, wobei dabei nur die blockverteilten Taskpools bessere Ergebnisse erzielen können.



(a) SGI Altix4700 (b) IBM p690

Abbildung 6.12: Speedups der Quicksort-Applikation mit adaptiven Taskpools und einer Feldgröße von 100.000.000.

6.3.2 Quicksort

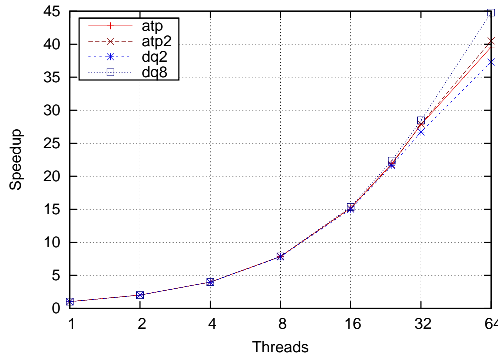
Bei der parallelen Quicksort-Applikation spielt die Nutzung der verfügbaren Parallelität eine wesentliche Rolle. Die adaptiven Taskpools sollen auch diese Anforderung genügend gut erfüllen. Abbildung 6.12 zeigt dazu die erzielten Resultate auf beiden Maschinen für die Sortierung von 100 Millionen Ganzzahlen. Der beste Speedup wird auf der SGI-Maschine von dem adaptiven Taskpool ohne private Bereiche bei 16 Threads mit $7,01$ erreicht, auf der IBM-Maschine erreicht dieselbe Implementierung einen Speedup von $6,97$ bei 32 Threads. Der adaptive Taskpool mit privaten Bereichen ist aber nur unwesentlich langsamer. Der verteilte Taskpool ohne Gruppierung erreicht ebenfalls gute Speedups.

Dagegen erzielt der blockverteilte Taskpool, wie erwartet, keine gute Performance. Auf der SGI-Maschine wird nur ein Speedup von $4,1$ erreicht, während auf der IBM-Maschine sogar nur ein Speedup von $2,74$ erreicht wird. Schon bei den vorherigen Untersuchungen verhinderte die Blockbildung eine bessere Skalierbarkeit, da die anfangs nur wenigen Tasks nur von einem einzigen Thread ausgeführt werden können. Die adaptiven Taskpools können diese Situation dagegen sehr gut handhaben.

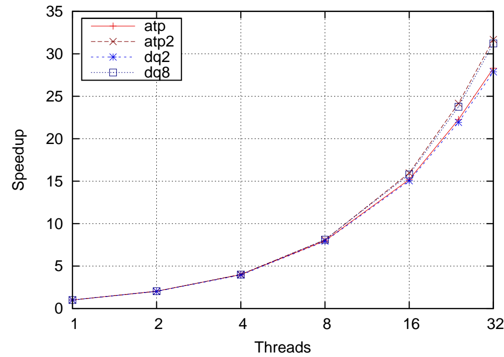
6.3.3 Ray-Tracing

Das Ray-Tracing ist sehr gut parallelisierbar und hat schon zuvor in Abschnitt 4.4.4 hohe Speedups erzielt, auch aufgrund der geringen Datenmodifikation innerhalb der Applikation. Allerdings erzeugt die vorliegende Ray-Tracing-Implementierung keine neuen Tasks zur Laufzeit und da die initial erzeugten Tasks auf alle Threads gleichmäßig verteilt sind, ist eine ungleiche Lastverteilung nicht zu erwarten. Daher sind die Verbesserungen durch die adaptiven Taskpools eventuell weniger stark.

Abbildung 6.13 zeigt die Ergebnisse auf den beiden betrachteten Maschinen. Die verschiedenen Implementierungen liegen nahe beieinander und erreichen nahezu linearen Speedup. Dennoch ist auf der SGI-Maschine der Speedup des blockverteilten Taskpools mit $44,7$ bei 64 Threads noch relativ weit vom Idealspeedup entfernt. Die adaptiven Taskpools erreichen ähnliche Speedups von ungefähr 40, während der verteilte Taskpool ohne Gruppierung nur

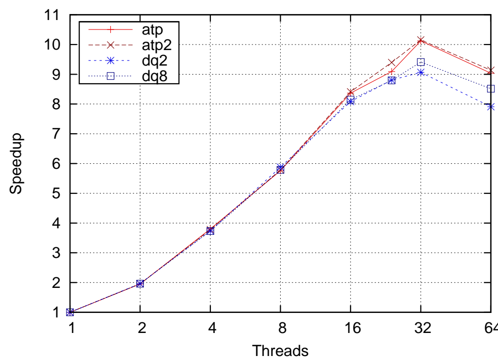


(a) SGI Altix4700

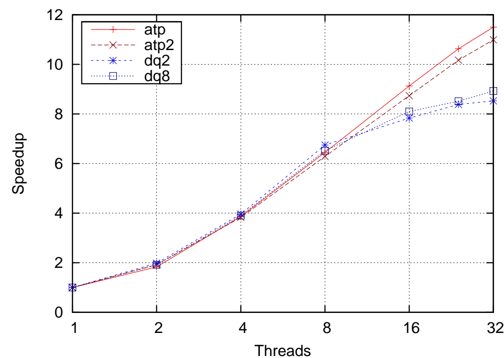


(b) IBM p690

Abbildung 6.13: Speedups der Ray-Tracing-Applikation mit adaptiven Taskpools für die Szene „car“ (1024x1024).



(a) SGI Altix4700



(b) IBM p690

Abbildung 6.14: Speedups der hierarchischen Radiosity-Applikation mit adaptiven Taskpools für die Szene „room11“.

einen Speedup von $37,3$ erreicht. Auf der IBM-Maschine erreicht der blockverteilte Taskpool einen Speedup von $31,2$. Allerdings ist hier der adaptive Taskpool mit privaten Bereichen mit einem Speedup von $31,7$ noch etwas schneller. Die adaptiven Taskpools erzielen also auch bei dieser Applikation gute Ergebnisse und sind nicht langsamer als konventionelle Taskpools, auch wenn aufgrund des Aufbaus der Applikation keine großen Verbesserungen durch den adaptiven Ansatz erzielt werden können.

6.3.4 Hierarchisches Radiosity

Das hierarchische Radiosity ist die komplexeste taskbasierte Applikation unter den betrachteten Applikationen. Vier verschiedene Tasktypen werden mit einem hohen Grad von Irregularität während der Laufzeit erzeugt.

Abbildung 6.14 zeigt die Ergebnisse der Laufzeittests auf beiden Maschinen. Auf der SGI-Maschine bricht die Skalierbarkeit bei mehr als 32 Threads für alle Taskpool-Implementie-

rungen ein. Der maximale Speedup von 10,2 wird bei 32 Threads von beiden adaptiven Taskpools erreicht, während die verteilten und blockverteilten Varianten einen etwas geringeren Speedup erreichen. Auf dem IBM-System sind die Ergebnisse nur etwas besser, allerdings immer noch weit vom Idealspeedup entfernt. Der beste Speedup wird vom adaptiven Taskpool ohne privaten Bereich mit 11,5 bei 32 Threads erreicht. Auf dieser Maschine brechen die verteilten und blockverteilten Taskpools dagegen auch schon bei 16 Threads ein und erreichen einen deutlich geringeren Speedup.

Die adaptiven Taskpools können also diese Applikation mit komplexer Taskerzeugung besser handhaben als die konventionellen Implementierungen. Allerdings zeigen die Ergebnisse insbesondere auf der SGI-Maschine, dass bei dieser Applikation ein Skalierbarkeitsproblem existiert. Daher wird diese Applikation im folgenden Kapitel als Beispiel für eine Profiling-Untersuchung benutzt, um die vorhandenen Probleme zu finden und zu beheben.

6.4 Zusammenfassung

Die adaptive Datenstruktur zur Speicherung der ausführungsbereiten Tasks ermöglicht die effiziente Handhabung einer großen Anzahl ausführungsbereiter Tasks. Laufzeittests haben gezeigt, dass der Overhead gering ist und durch effizientere Lastbalancierung bessere Ergebnisse erzielt werden als mit normalen Listen. Mit nur wenigen Operationen können die Tasks eines Threads auf die anderen Threads verteilt werden, sodass nicht notwendigerweise eine gute Anfangsverteilung vorliegen muss. Die Experimente haben gezeigt, dass die Performance der adaptiven Taskpools zwischen den verteilten Taskpools ohne Gruppierung und den blockverteilten Taskpools liegt, wobei aufgrund der effizienteren Lastbalancierung teilweise auch bessere Ergebnisse erzielt wurden als mit den blockverteilten Taskpools. Zudem können die adaptiven Taskpools auch Situationen handhaben, bei denen die blockverteilten Taskpools Probleme zeigen.

Kapitel 7

Analyse taskbasierter Applikationen mittels Task-Profiling

Bei der Laufzeitanalyse einer taskbasierten Applikation zeigt sich oft, dass die erreichten Speedups nicht den Erwartungen entsprechen. Um eventuelle Engpässe im Programm zu identifizieren, sind Hilfsmittel notwendig, die speziell für die taskbasierte Abarbeitung optimiert sind. Schlechte Laufzeitergebnisse können viele Ursachen haben. Probleme können in der Taskpool-Implementierung liegen, sodass Verbesserungen dort auch zu einer Laufzeitverbesserung der Applikation führen. Sind Skalierbarkeitsprobleme jedoch in der Applikation vorhanden, kann eine optimierte Taskpool-Implementierung keine Verbesserung bewirken.

Das KOALA-Framework enthält eine Komponente, mit der eine detaillierte Analyse der Taskstruktur einer Applikation möglich ist. Diese Komponente und die zugrunde liegenden Methoden werden in diesem Kapitel näher beschrieben. Zur Nutzung dieser Komponente sind keine Änderungen an der Applikation und der Taskpool-Implementierung notwendig. Die vorgestellte Profiling-Komponente kann insbesondere für ein paralleles Profiling eingesetzt werden, da viele Laufzeitprobleme erst durch die parallele Taskabarbeitung entstehen, wenn z. B. Locks die Skalierbarkeit limitieren. Mit Hilfe von gemessenen Laufzeitparametern einzelner Tasks lassen sich Aussagen über die Güte der Taskimplementierung machen und Vorhersagen treffen, wie die gegebene Applikation mit einer größeren Anzahl von Threads skalieren wird.

Es ist dabei generell schwierig, Aussagen über die Ausführungszeit eines taskbasierten Programms zu machen, da dies von Hardware-Parametern wie der CPU-Geschwindigkeit oder der Speicherbandbreite/-latenz genauso abhängt, wie von Eingabeparametern, die z. B. Schleifengrenzen bestimmen. Ein weiterer, schwer vorherzusagender Faktor bei der parallelen Abarbeitung sind Wartezeiten innerhalb des Taskpool-Frameworks. Für Aussagen über die Skalierbarkeit ist es auch wichtig zu wissen, ob zu jedem Zeitpunkt der Abarbeitung genügend Tasks verfügbar sind oder ob Änderungen in der Taskstruktur der Applikation notwendig sind. Daher sind genaue Informationen über die taskbasierte Abarbeitung hilfreich bei der Bewertung der parallelen Applikation und deren Nutzung der einzelnen Framework-Komponenten.

Bei der Betrachtung realer taskparalleler Applikationen können diese bezüglich ihrer parallelen Abarbeitung in folgende Klassen eingeordnet werden:

1. Applikationen, die nahezu ideal skalieren,
2. Applikationen, die nur bis zu einer bestimmten Anzahl von Prozessoren skalieren und

3. Applikation, die überhaupt nicht skalieren.

Die Grenzen der Klassen können nicht genau festgelegt werden und viele Faktoren spielen für die Einordnung eine Rolle. Selbst bei Applikationen der ersten Klasse können durch Task-Profiling Informationen gewonnen werden, die zwar keine unmittelbaren Verbesserungen der Ausführungszeit aufzeigen, aber durchaus Aussagen über die mögliche Skalierbarkeit auf anderen, eventuell größeren Parallelrechnern ermöglichen. Es können verdeckte Flaschenhälse bestimmt werden, die auf der getesteten Maschine keinen großen Einfluss haben, aber auf anderen Maschinen relevant sein könnten.

Interessanter sind dennoch die Applikationen der Klassen 2 und 3. Wenn eine Applikation nicht gut skaliert, muss die Ursache dafür gefunden und nach Möglichkeit beseitigt werden. Es kann dabei Probleme innerhalb der Applikation wie Zugriffskonflikte auf gemeinsame Daten oder eine ungeeignete Taskstruktur, durch z. B. zu kleine oder zu wenige Tasks, geben. Die Taskpool-Implementierung kann jedoch auch eine bessere Performance verhindern, wenn sie die gegebene Taskstruktur nicht effizient handhaben kann. Somit ist zu klären, welche Änderungen jeweils nötig sind, um die Skalierbarkeit zu verbessern.

Um Informationen zur taskparallelen Abarbeitung zu gewinnen, wird eine eigenständige Framework-Komponente eingesetzt, die die Messung einzelner Taskparameter mit geringem Overhead erlaubt. Dies geschieht sowohl für die Applikation als auch die Taskpool-Implementierung im Backend transparent, sodass dort keine Änderungen nötig sind.

Das Profiling ist eine bekannte Technik zur Analyse der Performance einer Applikation. Moderne Compiler unterstützen die automatische Instrumentierung durch Hinzufügen geeigneter Profiling-Operationen, die jedoch einen erheblichen zusätzlichen Aufwand bedeuten. Werkzeuge wie *VAMPIR* [18] erlauben die Analyse der so gesammelten Informationen.

Dagegen ist die in [87] präsentierte Methode nicht auf Codeinstrumentierung angewiesen, sondern kann direkt Binärcode verwenden und die Laufzeit vorhersagen. Bei der Methode werden Speicherzugriffe protokolliert, um daraus die Laufzeit für eine größere Eingabe vorherzusagen. Dabei ist man aber auf eine sequenzielle Ausführung beschränkt. In [67] wird dagegen versucht, die Performance einer Applikation bei der parallelen Ausführung auch auf zukünftigen Architekturen vorherzusagen, allerdings wird dafür eine detaillierte Analyse des Programmcodes der betrachteten Applikation benötigt. Es gibt auch Versuche, die Laufzeit anhand von Untersuchungen des Quellcodes vorherzusagen. In [73] wird beispielsweise die Laufzeit von MPI-Operationen vorhergesagt, um darauf aufbauend die Laufzeit eines kompletten MPI-Programms bestimmen zu können.

Um eine Applikation analysieren und Engstellen erkennen zu können, wird in [38] ein ähnlicher Ansatz wie in dieser Arbeit verfolgt. Mit Hilfe eines Simulators werden Speicherzugriffe erfasst, um daraus Vorschläge für Verbesserungen ableiten zu können. Bei dem in dieser Arbeit vorgestellten Ansatz wird auf den Overhead eines Simulators und auf Codeinstrumentierung verzichtet. Außerdem erfasst dieser Ansatz auch Probleme, die durch die parallele Abarbeitung und die damit eventuell verbundene gegenseitige Blockierung entstehen. In [86] wird weiterhin eine Methode vorgestellt, mit der parallele Applikationen mit *TAU* geprofilet werden können. Man kann damit zwar auch verschiedene Ausführungsphasen getrennt erfassen, allerdings wird instrumentierter Code benötigt.

7.1 Profiling-Komponente im Frontend

Die hier vorgestellte Profiling-Komponente wird zwischen der Applikationsschicht und der Taskpoolschicht eingesetzt, um unabhängig von beiden Schichten Laufzeitparameter zu bestimmen. Dazu wird im Folgenden beschrieben, wie die Taskabarbeitungsschleife in der Frontend-Funktion *tp_run* (vgl. Abschnitt 5.6.1) modifiziert wird, um u. a. Laufzeiten einzelner Tasks messen zu können. Mit diesen Informationen ist es möglich, die taskbasierte Abarbeitung zu bewerten und kritische Tasks zu identifizieren. Die Applikation und der Taskpool können unverändert eingesetzt werden, eine Code-Instrumentierung oder die Nutzung eines Simulators ist also nicht nötig.

7.1.1 Begriffsklärung

Die Berechnungen und damit die Ausführungszeiten im Taskpool-Framework lassen sich in zwei Klassen unterteilen. Auf der einen Seite sind die Berechnungen der Applikation zu betrachten. Dies liegt außerhalb des Verantwortungsbereichs des Taskpool-Frameworks. Da bei taskbasierten Applikationen diese Berechnungen in Tasks gekapselt sind, wird im weiteren Verlauf die Ausführungszeit der jeweiligen Tasks betrachtet. Deren Summe beschreibt die verbrauchte Zeit im *Applikationskontext*.

Im Einzelnen bezeichnet die *Taskgröße* ts eines Tasks die tatsächliche Ausführungszeit der mit dem Task verbundenen Berechnungen. Die Taskgröße $ts(T)$ eines Tasks T sei definiert durch die Differenz der Startzeit und der Endzeit des Tasks:

$$ts(T) = t_{end}(T) - t_{begin}(T).$$

Die Taskgröße ist somit abhängig von der Ausführungsplattform.

Darüber hinaus wird Rechenzeit innerhalb des Taskpool-Frameworks verbraucht (*Taskpoolkontext*), da nach Beendigung eines Tasks ein neuer Task zur Ausführung ausgewählt werden muss. Dies umfasst nicht nur die Verwaltungszeit, um entsprechend der jeweiligen Implementierung des Taskpool-Backends neue Tasks zu finden, sondern in dieser Zeit können auch Wartezeiten entstehen, wenn z. B. keine Tasks verfügbar sind oder Lock-Mechanismen zu einer Unterbrechung führen. Daher wird hier die Zeit zwischen Beendigung eines Tasks und dem Beginn der Abarbeitung eines neuen Tasks als *Wartezeit* wt bezeichnet. Sei T_{prev} ein abgeschlossener Task und T_{next} ein zur Ausführung ausgewählter Task, dann wird die Wartezeit entsprechend berechnet mit

$$wt = t_{begin}(T_{next}) - t_{end}(T_{prev}).$$

Da diese Zeit zwischen zwei Tasks im Taskpool gemessen wird, ist sie nicht unmittelbar einem einzelnen Task zugeordnet. Um eine spätere Analyse zu vereinfachen, es ist dennoch sinnvoll, diese Wartezeit ähnlich der Taskgröße mit einem Task zu verbinden.

Die Wartezeit kann entweder

(a) dem vorherigen Task T_{prev},

(b) dem T_{next} erzeugenden Task oder

(c) dem darauf folgenden Task T_{next}

zugeordnet werden. Der zuvor verarbeitete Task ist nicht direkt für eine anschließende Wartephase verantwortlich, da in der gesamten Applikation keine Tasks zur Verfügung stehen. Die Zuordnung zu dem Task, der den neuen Task zu einem früheren Zeitpunkt erzeugt hat, ist sinnvoller. Der erzeugende Task hätte entsprechend den neuen Task früher erzeugen müssen, um die Wartezeit zu reduzieren. Allerdings muss dazu zu jedem Task noch der erzeugende Task zumindest als Zeiger gespeichert werden. Um den mit dem Profiling verbundenen Overhead gering zu halten, wird diese Zuordnung daher nicht verwendet. Die gemessene Wartezeit wird dem neu auszuführenden Task zu geordnet, da der Task zwar noch nicht ausgeführt wurde, aber die Programmstelle, an der der Task erzeugt wurde, durchaus nachvollziehbar ist und so Rückschlüsse über eine zu geringe oder zu späte Taskerzeugung erlaubt. Im Kontext der Analyse genügt die Information, dass auf einen bestimmten Tasktypen besonders häufig gewartet wurde, er also in zu geringer Zahl oder zu spät erzeugt wurde. Dies kann mit der Zuordnung zum darauf folgenden Task bestimmt werden.

7.1.2 Implementierung der Komponente

Zur Erfassung der Profiling-Informationen werden die jeweils gemessenen Wartezeiten und Taskgrößen in den jeweiligen Kontexten protokolliert. Die Frontend-Komponente ist dafür gut geeignet, da sie als Schnittstelle zwischen der Applikation und der eigentlichen Taskpool-Implementierung dient. Daher wurde eine neue Frontend-Implementierung realisiert, das die nötigen Informationen in einer Datenstruktur sammelt und am Ende der Ausführung protokolliert. Die Profiling-Komponente kann durch Benutzung des Arguments

```
./configure --enable-prof=level1-hist
```

des *configure*-Skripts aktiviert werden. Die Implementierung greift auf die Schnittstelle zur Hardware-Zeitmessung (vgl. Abschnitt 5.3.3) zurück, um am entsprechenden Messpunkt mit wenig Overhead einen Zeitstempel zu erhalten. Abbildung 7.1 zeigt dazu den schematischen Aufbau. An der Stelle *X1* (Zeile 1) findet die Zeitmessung zu Beginn der Wartezeit im Taskpool statt. Wurde ein Task zur Ausführung gefunden, wird das Ende der Wartezeit an Position *X2* (Zeile 11) festgehalten und in einer geeigneten Datenstruktur abgespeichert. Unmittelbar vor der folgenden Ausführung wird erneut ein Zeitstempel erfasst (*X3* in Zeile 15), um nach Ende des Tasks dessen Laufzeit und somit dessen Taskgröße festzustellen (*X4* in Zeile 17). Dieser Messwert wird wiederum abgespeichert und anschließend beginnt die nächste Wartephase, sodass die entsprechende Anfangszeit in *X5* (Zeile 21) erfasst wird.

Die Zeitmessung erfolgt nach Möglichkeit durch Auslesen der Hardware-Zeitregister, sodass der Overhead dafür sehr gering ist. Wenn diese Operationen auf der Zielplattform nicht verfügbar sind, kann auf die Benutzung von *gettimeofday* mit entsprechend höherem Overhead und verringerter zeitlicher Auflösung zurückgegriffen werden (vgl. Kapitel 5.3.3).

Jeder taskausführende Thread führt die Abarbeitungsschleife aus Abbildung 7.1 aus, sodass die Messdaten parallel von allen Threads gesammelt werden. Wie in Kapitel 5.3.3 beschrieben, können dabei Probleme bei der Genauigkeit und Zuverlässigkeit der Zeitmessung mit Hardware-Unterstützung auftreten, wenn z. B. ein Thread vom Betriebssystem von einem Prozessor auf einen anderen verschoben wird. Die genannten Probleme sind hier

```
1   X1
2   for (;;) {
3       ret_val = tp_intern_get( ID, &function , &argument );
4       if ( ret_val == TP_NO_TASK ) {
5           WAIT_OR_EXIT;
6           continue;
7       } else if ( ret_val == TP_GOT_TASK_WITH_NEW_ONES &&
8                   tp_ready > 0 ) {
9           pthread_cond_signal( ... );
10      }
11      X2
12
13      store_waittime();
14
15      X3
16      (*(function))( ID, argument );
17      X4
18
19      store_tasksize();
20
21      X5
22      tp_free_arg( ID, argument );
23  }
```

Abbildung 7.1: Taskabarbeitungsschleife im Taskpool-Frontend mit markierten Stellen zur Zeitmessung.

jedoch kaum relevant, da einzelne Ausreißer aufgrund der beschriebenen Situationen für die statistische Auswertung kaum einen Einfluss haben.

Neben der gerade skizzierten Implementierung „level1-hist“, die die Datenerfassung entsprechend dem im folgenden Abschnitt beschriebenen Verfahren durchführt, gibt es noch eine Variante „level1-full“, welche die Messdaten der Reihe nach im Speicher protokolliert und am Ende ausgibt. Diese Implementierung erlaubt zwar ein detailliertere Analyse, allerdings wird sehr viel Speicher verbraucht und die erzeugte Protokolldatei ist sehr groß. Gerade für die Analyse feingranularer Applikationen mit entsprechend vielen Tasks fallen so viele Messdaten an, sodass sich die Implementierung „level1-hist“ als praktikabler herausgestellt hat.

7.1.3 Logarithmisches Histogramm

Für die Analyse der Taskgrößen mit Hilfe der Profiling-Informationen hat sich die Benutzung eines Histogramms als vorteilhaft herausgestellt, das die einzelnen Vorkommen verschiedener Taskgrößen bzw. Wartezeiten zählt. So kann bestimmt werden, wie häufig feingranulare Tasks in der Applikation vorgekommen sind oder ob beispielsweise öfters relativ lange auf neue Tasks gewartet werden musste. Das Histogramm basiert auf logarithmische Abständen. Dies ist notwendig, da kleine Zeitunterschiede bei geringen absoluten Messwerten für die Analyse wichtiger sind als ebenso kleine Zeitunterschiede bei sehr langen Laufzeiten. Deshalb wurden die Taskgrößen bzw. Wartezeiten in logarithmische Intervalle unterteilt, sodass alle gemessenen Werte zu einem entsprechenden Intervall gezählt werden. Somit wird die verfügbare zeitliche Auflösung voll genutzt, um kurze Zeitdifferenzen festzustellen, während bei langen Ausführungszeiten bzw. Wartezeiten die zeitliche Auflösung

entsprechend reduziert wird.

In der Profiling-Komponente wird also für jedes Zeitintervall gezählt, wie oft Tasks entsprechend lange gerechnet haben bzw. wie oft entsprechend lange gewartet werden musste. Für die Auswertung ist dies völlig ausreichend, da für kurze Zeiten exakte Daten notwendig sind, für längere Zeiten aber gröbere Daten ausreichen. So ist es beispielsweise interessant, ob ein Thread häufig $0-1\mu s$ warten musste, ob $5\mu s$ immer noch häufig auftrat, aber ob der Thread einmal $10005\mu s$ oder $10010\mu s$ warten musste, ist weniger relevant als die Aussage, dass überhaupt einmal $10000\mu s$ gewartet werden musste.

Berechnung der Intervalle und Protokollierung der Messwerte

Die Einteilung der einzelnen Intervalle basiert auf Stützstellen s_i mit $s_i = 10^{f(i)}$ wobei f eine vorgegebene, monoton steigende Abbildung $f : \mathbb{N} \rightarrow \mathbb{R}^+$ ist. Es ist sinnvoll, eine maximale Zeit festzulegen, sodass alle größeren Messwerte in das letzte Intervall fallen. So ist es nicht notwendig, bei der Betrachtung feingranularer Tasks solche Tasks zu unterscheiden, die Minuten oder Stunden rechnen, da eventuelle Verwaltungszeiten selbst im Mikrosekundenbereich dann nicht mehr relevant sind. Entsprechend sind sehr lange Wartezeiten ein Indiz für ein Skalierbarkeitsproblem, aber noch längere Wartezeiten müssen dann nicht mehr zusätzlich unterschieden werden. Somit kann ein i_{max} bestimmt werden:

$$0 \leq f(i) \leq f(i_{max}) = \lceil \log_{10}(\text{Messobergrenze}) \rceil$$

Die Abbildung f für die Berechnung der eigentlichen Stützstellen kann z. B. $f(i) = i$ sein, allerdings können ganzzahlige Schrittweiten zu grob sein. Für die folgenden Untersuchungen wird daher eine Reihe mit Zehntelschritten benutzt, also $f(i) = \frac{i}{10}$.

Die Intervalle bestimmen sich somit wie folgt:

$$\begin{aligned} I_0 &= [0; s_1) \\ I_k &= [s_k; s_{k+1}) \, \forall k, 1 \leq k < i_{max} - 1 \\ I_{i_{max}-1} &= [s_{i_{max}-1}; s_{i_{max}}]. \end{aligned}$$

Bei der Verwendung einer konstanten Schrittweite t kann das zu einem Messwert v entsprechende Intervall I_m berechnet werden mit:

$$m = \lfloor \log_{10}(v) / t \rfloor \, . \tag{7.1}$$

Anmerkung: Bei der Verwendung von Zehntelschritten wie in der weiteren Untersuchung sind zwischen 10^0 und 10^1 zehn mögliche Intervalle vorgesehen, allerdings sind durch die logarithmische Berechnung bei ganzzahliger Betrachtung einige Grenzen mehrfach enthalten. Da dieser Bereich jedoch besonders wichtig ist, werden die ersten zehn Intervalle entsprechend modifiziert, sodass disjunkte Bereiche mit Abstand 1 verwendet werden. Alle folgenden Intervalle entsprechen der obigen Berechnung.

Besonders bei einer großen Anzahl feingranularer Tasks mit vielen Threads fallen entsprechend große Menge von Daten an. Da sich die nachträgliche Analyse bei besonders großen Eingabedateien verschiedener Applikationen als kaum noch handhabbar gezeigt hat, sollen

die Informationen soweit möglich direkt bei der Erfassung vorsortiert werden. Für die dynamische Lastbalancierung sind jedoch insbesondere solche Tasks relevant, die nicht sehr lange laufen. Setzt man eine Obergrenze von z. B. einer Sekunde, so gibt es bei Mikrosekundenauflösung 1.000.000 verschiedene Messwerte, die sich bei Benutzung von Zehntelschritten auf nur 60 Intervalle aufteilen. Mit Hilfe der Gleichung (7.1) unter Benutzung der Logarithmusfunktion kann das Intervall bestimmt werden. Aufgrund der geringen Anzahl von Intervallen hat sich jedoch die binäre Suche nach dem Intervall als schneller herausgestellt. In einem Testlauf wurde festgestellt, dass eine binäre Suche mindestens um den Faktor 10 schneller ist als die Berechnung mit *log* (aus „math.h“). Auch bei Benutzung des Intel-Compilers statt des GNU-Compilers ist die binäre Suche immer noch um den Faktor 3-4 schneller gewesen.

Daher wird direkt bei der Bestimmung der Taskgröße bzw. der Wartezeit, wie in Abbildung 7.1 beschrieben, mittels der binären Suche das entsprechende Intervall bestimmt und in einem Feld fester Größe bezüglich der Taskfunktion aufsummiert. Nach Beendigung der Berechnung werden dann die gesammelten Daten ausgegeben.

7.2 Auswertung der Profiling-Informationen

Die Taskgrößen, also die Ausführungszeiten der einzelnen Tasks, repräsentieren die eigentlichen Berechnungen der Applikation. Je größer die gemessenen Taskgrößen sind, desto mehr Berechnungen wurden innerhalb der einzelnen Tasks ausgeführt. Mit diesem Messwert lassen sich also Aussagen über die Taskgranularität der betrachteten Applikation treffen. Darüber hinaus ist es jedoch auch möglich, Informationen über mögliche Engstellen bei der parallelen Abarbeitung zu erhalten. Sind z. B. die Berechnungen der einzelnen Tasks unabhängig von der Anzahl der ausführenden Threads, dann sollte die Taskausführungszeit auch bei steigender Anzahl von Threads entsprechend konstant sein. Steigt die gemessene Zeit jedoch systematisch an, ist dies ein Hinweis auf Skalierbarkeitsprobleme innerhalb der Tasks, beispielsweise aufgrund von Wechselwirkungen mit anderen Tasks. Gründe dafür können beispielsweise eine erhöhte Anzahl von Cache-Fehlzugriffen sein, erhöhte Speicherzugriffszeiten aufgrund von Zugriffen auf entfernten Speicher oder eine erhöhte Zeit für Lock-Operationen. Da für jeden Task neben der Ausführungszeit auch der ausführende Thread protokolliert wird, lässt sich dies in der anschließenden Analyse detailliert für jeden einzelnen Thread untersuchen.

Die zweite Messgröße ist die Wartezeit im Taskpool-Kontext. Diese Zeit enthält sowohl die benötigte Zeit für die Verwaltung der Tasks, also den Overhead der Taskpool-Implementierung, als auch Wartezeiten, falls z. B. bei einem Entnahmeversuch keine Tasks verfügbar sind. Da diese Zeit nicht für die eigentlichen Berechnungen verwendet wird, wird diese Zeit als Overhead betrachtet, der möglichst gering sein sollte. Ein Anstieg dieser Zeit bei der Benutzung zusätzlicher Threads weist ebenso auf ein Skalierbarkeitsproblem hin. Dies kann in einer schlechten Taskpool-Implementierung begründet sein, die die Tasks der Applikation nicht effizient verwalten kann (weil diese z. B. zu klein sind). Entstehen Wartezeiten, weil keine Tasks zur Ausführung bereitstehen, dann liegt das Problem bei der Applikation, die entsprechend mehr Tasks erzeugen sollte, damit alle Threads mit Berechnungen beschäftigt sind. Da der Overhead der Taskpool-Implementierung z. B. durch Untersuchungen mit der

synthetischen Applikation relativ genau bekannt ist, können Informationen über die Wartezeit benutzt werden, um Probleme innerhalb der Applikation zu finden. Die gemessenen Wartezeiten lassen sich wiederum den einzelnen Threads zuordnen, da dies entsprechend mitprotokolliert wird.

Die Ausführung einer taskbasierten Applikation mit dem Profiling-Taskpool-Frontend erzeugt eine große Menge von Messdaten der einzelnen Tasks, die entsprechend analysiert werden muss. Dabei lassen sich aus den Messdaten allgemeine Aussagen über die Abarbeitung treffen:

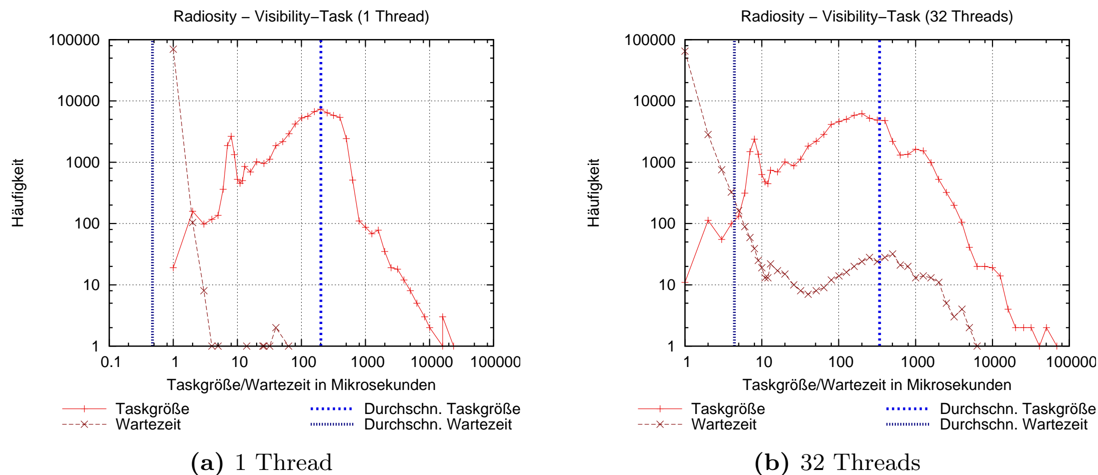
1. die Anzahl ausgeführter Tasks,
2. die Summe der Berechnungszeiten (also die gesamte Zeit in der Applikation),
3. die durchschnittliche Taskgröße,
4. die Summe der Wartezeiten (die gesamte Zeit außerhalb der Applikation),
5. die durchschnittliche Wartezeit und
6. die kleinsten und größten Wartezeiten und Taskgrößen.

Mit diesen allgemeinen Informationen lassen sich erste Schlüsse über die Applikation und deren Taskstruktur ziehen. Wenn die Anzahl der ausgeführten Tasks relativ gering ist, ist der Einfluss der Taskpool-Implementierung und entsprechend der verwendeten Lastbalancierung gering. Die durchschnittliche Taskgröße ist ein Indiz für den möglichen Einfluss des Taskpools auf die Performance. Werden beispielsweise im Durchschnitt nur relativ lang laufende Tasks bearbeitet, spielt der Overhead zur Suche von ausführungsbereiten Tasks keine große Rolle. Die Wartezeit muss als Overhead betrachtet werden, entsprechend deuten relativ hohe durchschnittliche Wartezeiten auf ein Problem innerhalb der Applikation hin, da nicht genügend Tasks zur parallelen Abarbeitung verfügbar waren.

Mit diesen Eckdaten sind zwar allgemeine Aussagen möglich, um Probleme zu identifizieren und Lösungen dazu vorzuschlagen, sind jedoch detaillierte Analysen notwendig. Deshalb werden Taskhistogramme wie in Abbildung 7.2 erzeugt, die für jede Taskgröße und Wartezeit deren Vorkommen auftragen. Mit Hilfe des Profiling-Frontends „level1-hist“ sind die Daten direkt so aufbereitet, dass eine logarithmische Achsendarstellung verwertbare Resultate liefert. Die y-Achse ist auch logarithmisch skaliert, da hier ebenfalls geringe Vorkommen bestimmter x-Werte relevanter sind als der genaue Wert großer Vorkommen. Zum Beispiel können schon wenige, aber große Wartezeiten auf ein mögliches Problem innerhalb der Applikation hinweisen.

Mit der zusätzlich dargestellten durchschnittlichen Taskgröße und Wartezeit lassen sich so bei der Gegenüberstellung zweier Taskhistogramme mit einer unterschiedlichen Anzahl benutzter Threads Rückschlüsse auf die taskbasierte Abarbeitung wie folgt schließen:

- Eine ähnliche Form der Taskgröße-Graphen für eine unterschiedliche Anzahl von Threads deutet auf eine geeignete Taskstruktur hin, bei der die Ausführungszeit eines Tasks nicht von der Anzahl der Threads abhängt. Verschiebt sich der Graph jedoch nach rechts, d. h. in Bereiche längerer Tasklaufzeiten, so zeigt dies Probleme innerhalb



(a) 1 Thread

(b) 32 Threads

Abbildung 7.2: Taskhistogramm für den Tasktyp „Visibility“ der Radiosity-Applikation auf der IBM-Maschine (Szene „largeroom“).

des betrachteten Tasktyps an. Ursachen dafür können z. B. Cache-Fehlzugriffe durch parallele Zugriffe mehrerer Threads auf gemeinsame Daten sein.

- Viele kleine Tasks, d. h. ein hohes Vorkommen von Tasks mit kurzen Laufzeiten im linken Teil des Histogramms, deuten auf einen hohen Overhead des Taskpool hin, da dieser besonders häufig zugegriffen wird. Eine schlechte Taskpool-Implementierung kann also problematisch für die Gesamtlaufzeit sein.
- Große Tasks (lange Laufzeiten im rechten Teil des Histogramms) bedeuten entsprechend einen geringen Einfluss der gewählten Taskpool-Implementierung. Allerdings könnte die Parallelität geringer sein, wenn die Berechnungen nur auf wenige große Tasks verteilt sind.
- Viele Tasks im mittleren Bereich weisen auf eine gute Balance zwischen dem Overhead im Taskpool durch zu kleine Tasks und erhöhter ungleicher Lastverteilung durch wenige große Tasks hin.
- Die Wartezeiten sollten generell so gering wie möglich sein, d. h. möglichst viel auf der linken Seite des Histogramms.
- Große Wartezeiten, auch wenn solche nicht häufig auftreten, weisen auf geringe Parallelität in der Applikation hin, da zu einigen Zeitpunkten manche Threads nicht mit Berechnungen beschäftigt waren und im Taskpool auf neue Tasks warten mussten.
- Die Kurve der Wartezeiten sollte signifikant unter der Kurve der Taskgrößen liegen, da Wartezeiten wesentlichen kleiner als Tasklaufzeiten sein sollten. Ansonsten wird mehr Zeit in der Taskverwaltung als mit den eigentlichen Berechnungen verbraucht.

Im Folgenden soll die Analyse anhand eines ausgewählten Tasktyps der Radiosity-Applikation beschrieben werden, die im folgenden Abschnitt 7.3.1 genauer betrachtet wird.

Neben der Profiling-Frontend-Komponente wird der verteilte Taskpool „DQ-PTH“ (*dq2*) mit der entsprechenden Pthreads-Lock-Komponente genutzt, da diese einfache Taskpool-Implementierung keine großen Einflüsse, z. B. durch etwaige interne Taskpufferung in privaten Bereichen, auf die Applikation ausübt. Die Abbildung 7.2 zeigt die Resultate für den „Visibility“-Tasktyp der Radiosity-Applikation, um die Sichtbarkeit von Dreiecken im Raum zu bestimmen (siehe dazu auch Kapitel 3.4). Bei Betrachtung der Ergebnisse für einen Thread (Abbildung 7.2a) zeigt sich, dass die Wartezeit der meisten Tasks eine Mikrosekunde beträgt (der Messwert 0 wird aufgrund der logarithmischen Darstellung zum Messwert 1 dazugezählt). Da bei der sequenziellen Abarbeitung keine Wartezeit aufgrund nicht verfügbarer Tasks auftreten kann, beschreibt diese Zeit also den reinen Taskpool-Overhead. Dagegen ist die durchschnittliche Taskgröße $201\,\mu s$ und es gibt nur einige hundert von über 70,000 Tasks mit einer Taskgröße $\leq 5\,\mu s$. Die durchschnittliche Wartezeit (mit Berücksichtigung der 0-Messwerte) liegt bei $\approx 0,5\,\mu s$, sodass die Taskgröße also ungefähr 400 mal größer ist als die Wartezeit bzw. der Taskpool-Overhead.

Für 32 Threads in Abbildung 7.2b zeigt sich, dass es bei der Taskgröße keine wesentliche Abhängigkeit von der Anzahl beteiligter Threads gibt, da der entsprechende Graph einen ähnlichen Verlauf hat. Das heißt also, dass die Tasks eine ähnlich lange Zeit zum Berechnen der zugewiesenen Aufgaben benötigen wie bei der sequenziellen Abarbeitung. Obwohl es eine leichte Verschiebung in den rechten Bereich des Histogramms gibt, ist dieser Tasktyp kaum durch Speicherbandbreite oder Cache-Fehlzugriffe beschränkt.

Die Wartezeiten dagegen steigen bei 32 Threads signifikant an. Während die durchschnittliche Wartezeit mit $4,4\,\mu s$ etwa zehn mal höher ist als bei der sequenziellen Abarbeitung, steigt die durchschnittliche Taskgröße gerade einmal von $201\,\mu s$ auf $340\,\mu s$. Einige Wartezeiten liegen schon im Millisekundenbereich und die größte Wartezeit ist mit $6319\,\mu s$ schon etwa ein Zehntel der Ausführungszeit des größten ausgeführten Tasks. Da allerdings die durchschnittliche Taskgröße immer noch rund 80 mal größer ist als die durchschnittliche Wartezeit, hat dieser Anstieg auf diesem System noch keinen wesentlichen Einfluss auf die Laufzeit der Applikation. Auf einem größerem oder schnelleren System kann die steigende Wartezeit jedoch die Skalierbarkeit einschränken.

Für diesen Tasktypen der Radiosity-Applikation kann daher folgender Schluss gezogen werden:

1. Die großen Taskgrößen im Vergleich zu den Wartezeiten weisen auch bei 32 Threads auf gute Skalierbarkeit hin.

2. Der Overhead der Taskpool-Implementierung ist bei diesem Tasktyp eher vernachlässigbar.

3. Die durchschnittliche Taskgröße steigt bei paralleler Abarbeitung mit 32 Threads etwas an (nicht ganz verdoppelt), während sich die durchschnittliche Wartezeit verzehnfacht hat. Ideale Skalierbarkeit insbesondere auf größeren Systemen kann nicht erwartet werden.

4. Die geringe Zahl sehr kleiner Tasks weist auf eine geeignete Taskstruktur hin, die feingranular genug ist, um gute Lastbalancierung zu realisieren, aber nicht zu klein, sodass der Taskpool-Overhead die Applikation behindern würde.

Im folgenden Abschnitt werden die anderen Tasktypen der Radiosity-Applikation und die anderen verfügbaren Applikationen betrachtet.

7.3 Fallbeispiele für die Applikationsanalyse

Da die Profiling-Komponente transparent zwischen der Applikation und der eigentlichen Taskpool-Implementierung liegt, können die verfügbaren taskbasierten Applikationen ohne Änderungen mit dieser Komponente analysiert werden. In diesem Abschnitt wird deshalb die Radiosity-Applikation genauer betrachtet, nachdem zuvor schon ein ausgewählter Tasktyp dieser Applikationen beschrieben wurde. Außerdem werden die Ergebnisse der anderen Applikationen dargestellt.

Die Profiling-Informationen wurden jeweils auf der IBM-Maschine mit 32 Power4+-Prozessoren bestimmt. Für die Lock-Komponente wurden Pthreads-Locks verwendet, da die Verwendung von Hardware-basierten Locks aufgrund der verwendeten „busy-loops“ die User-Zeit zusätzlich erhöht. Dies kann zu Verfälschungen der Profiling-Analyse führen. Wartezeiten, die aufgrund von Blockierungen bei der Anfrage eines Locks entstehen, werden jedoch sowohl bei Hardware-Locks als auch bei Pthreads-Locks mit gemessen und fließen entsprechend in die Analyse mit ein. Als Taskpool wird der verteilte Taskpool „DQ-PTH“ (*dq2*) genutzt.

7.3.1 Untersuchung der Radiosity-Applikation

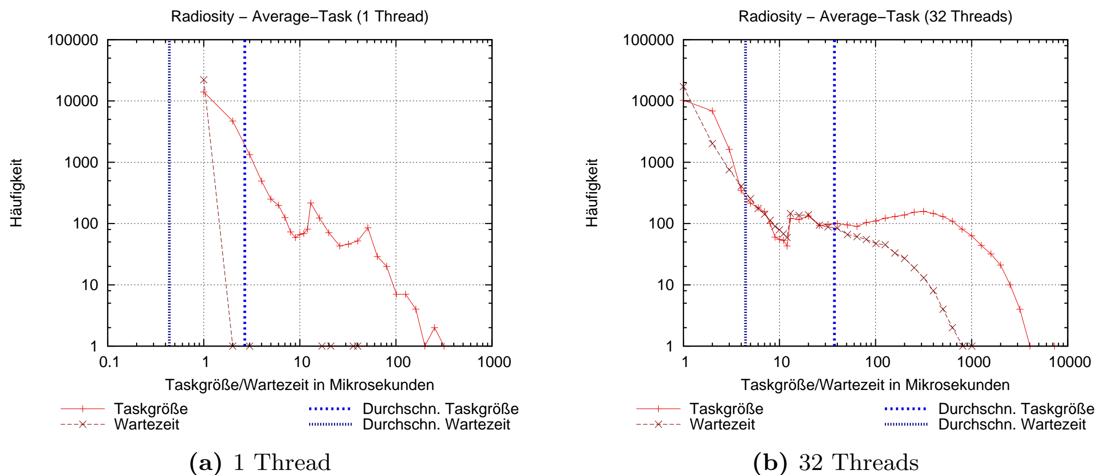
Die Radiosity-Applikation eignet sich durch die dynamische Taskerzeugung unterschiedlicher Tasktypen gut als Fallbeispiel für die Auswertung der Profiling-Informationen. Wie in Kapitel 3.4 beschrieben besteht die Applikation aus vier verschiedenen Tasktypen, die hier nochmals kurz zusammengefasst werden.

Zu Beginn der Abarbeitung werden die Oberflächen der betrachteten Szene mit „Refinement“-Tasks unterteilt, die entsprechend jeweils parallel abgearbeitet werden können. Die Berechnungen der Strahlenverfolgung erfolgt durch „Ray“-Tasks. Ein „Ray“-Task kann weitere „Ray“- und „Visibility“-Tasks erzeugen, wenn dies aufgrund von Fehlerschranken notwendig sein sollte. Der „Average“-Task führt am Ende der Berechnungen eine Nachbearbeitung der berechneten Strahlungswerte aus, um die finale Darstellung zu verbessern.

„Visibility“-Task. Dieser Tasktyp wurde im vorherigen Abschnitt betrachtet. Dabei wurden keine wesentlichen Probleme für die maximale Anzahl von 32 Prozessoren festgestellt.

„Average“-Task. Abbildung 7.3 zeigt das Taskhistogramm für den „Average“-Task, der in der Nachbearbeitungsphase die berechneten Radiosity-Werte normalisiert.

Der Großteil der Tasks ist sehr klein, weniger als $10\,\mu s$. Für einen Thread (Abbildung 7.3a) ist die durchschnittliche Taskgröße $2,7\,\mu s$ und nur einige wenige Tasks rechnen länger als $100\,\mu s$. Der Overhead des Taskpools ist aufgrund dieser sehr feingranularen Tasks verhältnismäßig hoch. Die Wartezeit, hier also der Overhead des Taskpools, ist durchschnittlich $0,4\,\mu s$ und somit schon bei der sequenziellen Ausführung ungefähr 16 Prozent der durchschnittlichen Taskgröße.



(a) 1 Thread **(b)** 32 Threads

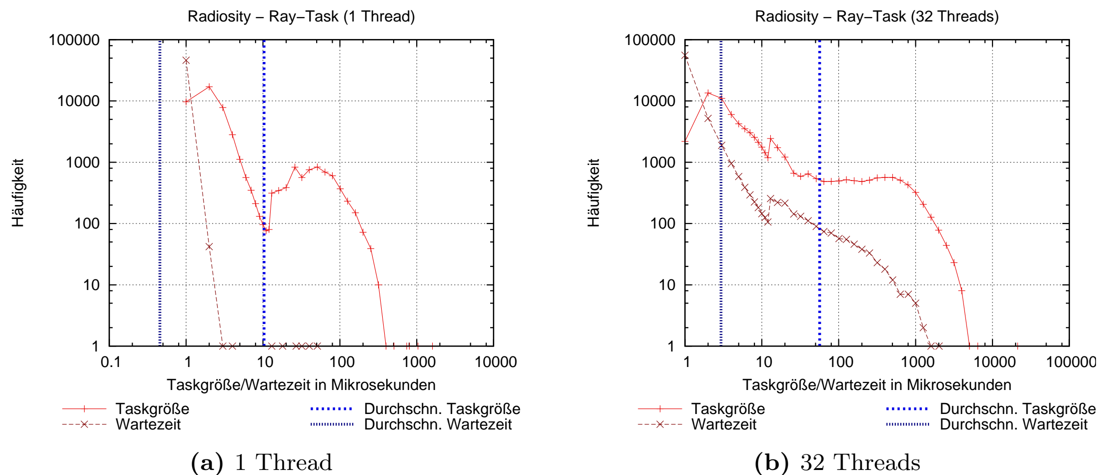
Abbildung 7.3: Taskhistogramm für den Tasktyp „Average“ der Radiosity-Applikation auf der IBM-Maschine (Szene „largeroom“).

Bei 32 Threads (Abbildung 7.3b) steigt wie erwartet die Wartezeit stark an, aber auch der Graph der Taskgröße hat sich stark verändert. Es gibt jetzt eine große Anzahl von Tasks mit einer Laufzeit größer als $100\,\mu s$ und die durchschnittliche Taskgröße ist mit $37,4\,\mu s$ fast 14 mal größer als bei der sequenziellen Abarbeitung, d. h. es gibt bei diesem Tasktyp ein Problem innerhalb der Taskstruktur, da die eigentlichen Berechnungen eines Tasks nicht von der Anzahl teilnehmender Threads abhängen. Dies kann auf Lock-Probleme, eingeschränkte Speicherbandbreite oder Cache-Fehlzugriffe hinweisen.

Die Wartezeit ist zehn mal größer ($4,4\mu s$) als bei der sequenziellen Abarbeitung, dieser Anstieg ist auch durch die relativ feingranularen Tasks zu erklären. Trotz des geringeren Anstiegs gegenüber der durchschnittlichen Taskgröße ist die durchschnittliche Wartezeit immer noch etwa 12 Prozent der durchschnittlichen Taskgröße. Zudem überlagert der Graph der Wartezeiten bei 32 Threads in weiten Teilen den Graph der Taskgrößen und es wurde relativ oft länger auf einen Task gewartet, als dann laut der durchschnittlichen Taskgröße zu rechnen war.

Somit kann man schließen, dass dieser Tasktyp nicht gut skaliert, da einerseits die Tasks recht klein sind und somit der Overhead der Taskpool-Implementierung entsprechend groß ist, und andererseits der signifikante Anstieg der Taskgröße auf ein Problem innerhalb des Tasktyps hinweist. Neben der Wahl einer effizienteren Taskpool-Implementierung muss hier die Taskstruktur der Applikation selbst verbessert werden, um die Skalierbarkeit zu erhöhen.

„Ray“-Task. Mit dem „Ray“-Task wird der eigentliche Strahlungsaustausch zwischen den Elementen der Szene berechnet. Bei sequenzieller Abarbeitung (Abbildung 7.4a) sind die Wartezeiten und somit der Taskpool-Overhead sehr gering. Es gibt sehr viele kleine Tasks (Laufzeit $\leq 10\,\mu s$), aber es gibt auch viele größere Tasks, sodass die durchschnittliche Taskgröße $10,3\,\mu s$ beträgt. Die durchschnittliche Wartezeit ist dagegen mit $0,5\,\mu s$ nur etwa



(a) 1 Thread **(b)** 32 Threads

Abbildung 7.4: Taskhistogramm für den Tasktyp „Ray“ der Radiosity-Applikation auf der IBM-Maschine (Szene „largeroom“).

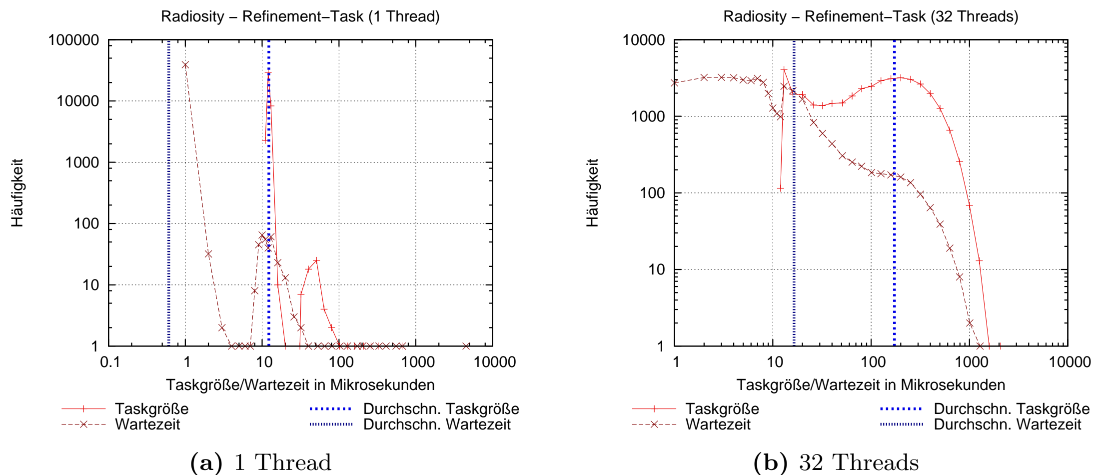
4 Prozent der Taskgröße.

Bei 32 Threads (Abbildung 7.4b) steigt die durchschnittliche Wartezeit auch aufgrund der vielen kleinen Tasks und dem damit verbundenen Taskpool-Overhead auf $2,9\,\mu s$ an. Auch die Taskgröße steigt fast um den Faktor $5,5$ auf $56,4\,\mu s$, und der Graph hat sich entsprechend leicht verschoben.

Zwar ist der Graph der Wartezeiten noch relativ deutlich unterhalb des Graphen der Taskgrößen, der Anstieg der Taskgröße deutet aber auf ein ähnliches Skalierbarkeitsproblem innerhalb des Tasktyps hin wie bei dem „Average“-Task. Eine bessere Taskpool-Implementierung würde hier dagegen einen nicht so großen Einfluss haben.

„Refinement“-Task. Der letzte zu betrachtende Tasktyp ist der „Refinement“-Task, der in der Startphase ausgeführt wird, um die Elemente der Eingabeszene geeignet zu unterteilen. Die Analyse der Profiling-Informationen lässt keine gute Skalierbarkeit erkennen. Bei sequenzieller Abarbeitung (Abbildung 7.5a) gibt es zwar keine Tasks, die kleiner als $11\,\mu s$ sind, allerdings auch nur wenige, die wesentlich länger laufen. Daher ist die durchschnittliche Taskgröße $12,2\,\mu s$. Die Wartezeiten sind zwar wie erwartet gering (durchschnittlich $0,6\,\mu s$), aber der Graph zeigt eine starke Häufung bei $10 - 50\,\mu s$ und einige Wartezeiten betragen bis zu $4500\,\mu s$. Diese gemessenen Zeiten beschreiben den Overhead der Taskpool-Implementierung zur Erzeugung der nötigen Datenstrukturen zur Speicherung der großen Anzahl von Tasks bei Beginn der Abarbeitung.

Bei 32 Threads (Abbildung 7.5b) haben sich beide Graphen signifikant verändert und weisen auf ein Skalierbarkeitsproblem hin. Die durchschnittliche Taskgröße ist mit $172,8\,\mu s$ mehr als 14 mal größer als bei der sequenziellen Ausführung und die durchschnittliche Wartezeit ist sogar fast 27 mal größer ($16,5\,\mu s$). Auch wenn es gegenüber den anderen betrachteten Tasks keine sehr kleinen Tasks gibt ($< 10\,\mu s$), so ist doch die Skalierbarkeit durch die hohen Wartezeiten und den starken Anstieg der Taskgröße stark limitiert. Hier muss



(a) 1 Thread

(b) 32 Threads

Abbildung 7.5: Taskhistogramm für den Tasktyp „Refinement“ der Radiosity-Applikation auf der IBM-Maschine (Szene „largeroom“).

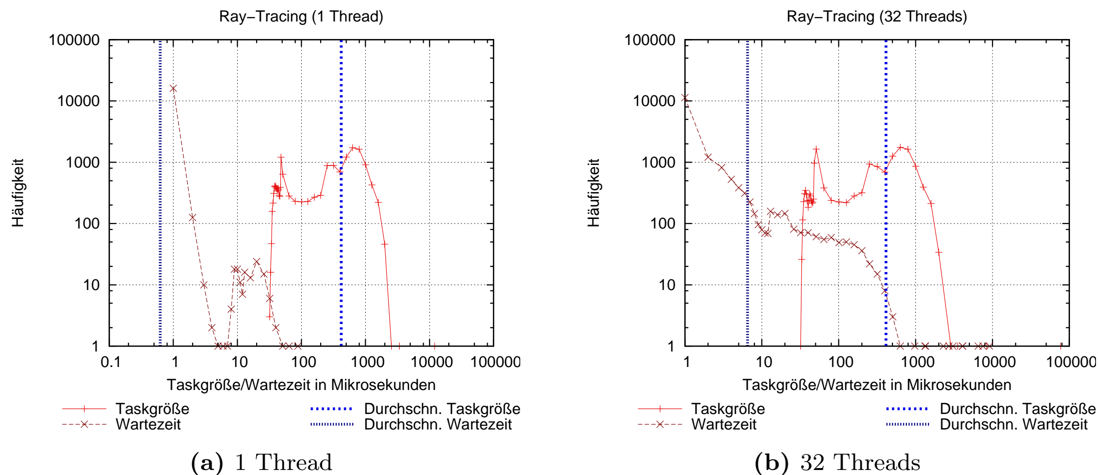
eine Veränderung innerhalb der Applikation vorgenommen werden, um bessere Laufzeiten zu erhalten.

Zusammenfassung der Histogramm-Ergebnisse. Mit Hilfe der detaillierten Informationen über die einzelnen Tasks können problematische Tasks identifiziert werden. Die Taskstruktur gibt auch Auskunft, ob eine verbesserte Taskpool-Implementierung die parallele Laufzeit verbessern kann.

Bei der Untersuchung der Radiosity-Applikation wurde festgestellt, dass hauptsächlich Probleme innerhalb der Applikation die Skalierbarkeit limitieren. Im Wesentlichen sollten hier der „Average“-Task und der „Refinement“-Task verbessert werden, da hier die größten Probleme festgestellt wurden. Während beim „Refinement“-Task Probleme innerhalb der Taskstruktur zu beheben sind, bremst beim „Average“-Task die zu feingranularen Tasks die parallele Abarbeitung. Als drittes kann der „Ray“-Task nach Möglichkeit verbessert werden, während der „Visibility“-Task schon relativ gut funktioniert. Im Abschnitt 7.6 wird dazu eine entsprechend verbesserte Variante umgesetzt.

7.3.2 Ray-Tracing

In den vorherigen Experimenten hat sich die Ray-Tracing-Applikation als gut skalierend herausgestellt. Dies wird durch die Taskhistogramme (Abbildung 7.6) weitgehend bestätigt. Die durchschnittliche Taskgröße ist bei sequenzieller Ausführung (Abbildung 7.6a) mit $418\,\mu s$ recht hoch, zudem es keine Tasks kleiner als $25\,\mu s$ gibt. Die Wartezeit ist gering, der Ausschlag bei ungefähr $10\,\mu s$ ist wieder auf die Initialisierung der Datenstrukturen innerhalb des Taskpools zurückzuführen, da bei dieser Applikation nur dieser eine Tasktyp existiert. Der Graph der Taskgrößen bei 32 Threads (Abbildung 7.6b) ist nahezu identisch zur sequenziellen Ausführung, entsprechend bleibt die durchschnittliche Taskgröße mit $412\,\mu s$ nahezu konstant. Die Berechnungen eines Tasks sind also nicht abhängig von



(a) 1 Thread **(b)** 32 Threads

Abbildung 7.6: Taskhistogramm der Ray-Tracing-Applikation auf der IBM-Maschine (Szene „car“ 512×512).

der Anzahl benutzter Threads und die parallele Ausführung behindert auch nicht jeweils andere Tasks. Es existieren also kaum Zugriffskonflikte.

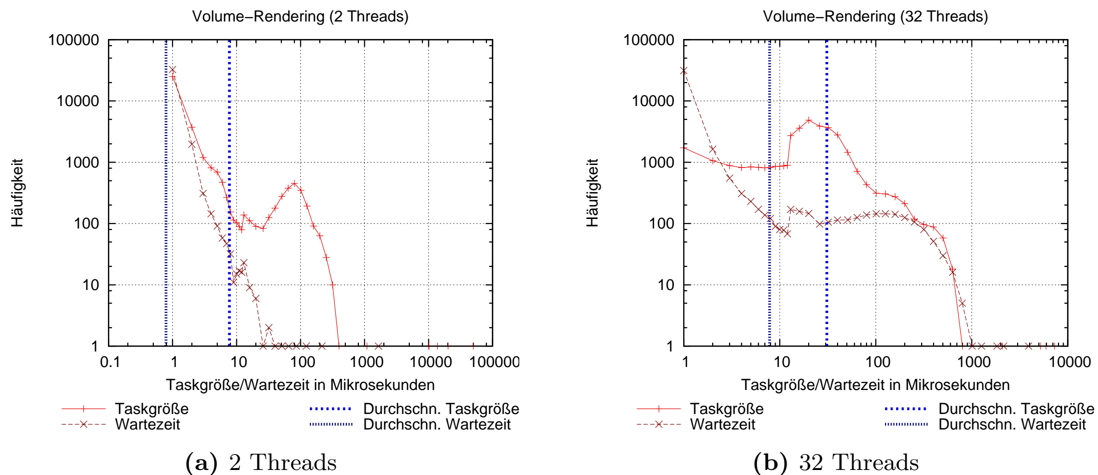
Die Wartezeiten zeigen jedoch einen signifikanten Anstieg von durchschnittlich $0,6\,\mu s$ bei sequenzieller Abarbeitung auf $6,5\,\mu s$ bei 32 Threads. Zwar ist die Mehrzahl der Wartezeiten deutlich kleiner als die Taskgrößen, so gibt es doch einige relativ lange Wartezeiten von bis zu $9165\,\mu s$.

Diese Applikation weist also eine gute Skalierbarkeit auf, da die Taskstruktur für eine effiziente Ausführung geeignet ist. Allerdings lassen die ansteigenden Wartezeiten eine schlechte Skalierung bei einer größeren Zahl von Prozessoren erwarten (tatsächlich wird bei 32 Threads ein etwas geringerer Speedup von 30-31 erreicht). Dieses Problem lässt sich aber auf Taskpool-Seite durch die Verwendung einer effizienteren Taskpool-Implementierung beheben.

7.3.3 Volume-Rendering

Die Volume-Rendering-Applikation konnte bei den vorhergegangenen Experimenten keine guten Speedups erreichen, was auf eine ungeeignete Taskstruktur hindeutet.

Die Taskhistogramme in Abbildung 7.7 weisen wie erwartet auf sehr viele besonders kleine Tasks hin, die einen entsprechenden Overhead bei der Verwaltung erzeugen. Bei zwei Threads (Abbildung 7.7a, die Implementierung nutzt bei sequenzieller Abarbeitung keine Taskpools) ist die durchschnittliche Taskgröße $7,8\,\mu s$, während die Wartezeit mit $0,8\,\mu s$ zwar klein ist, aber die benötigten Zeiten für die Erzeugung der internen Datenstrukturen im Taskpool-Framework sind deutlich sichtbar. Bei 32 Threads (Abbildung 7.7b) verschiebt sich der Graph der Taskgrößen deutlich in den rechten Bereich. Die durchschnittliche Taskgröße ist fast vier mal so groß ($30,9\,\mu s$) wie die Taskgröße bei der Ausführung mit zwei Threads. Auffällig ist, dass es deutlich weniger sehr kleine Tasks gibt. Die Anzahl ausführender Threads hat also Einfluss auf die Berechnungen, was auf Probleme innerhalb des



(a) 2 Threads

(b) 32 Threads

Abbildung 7.7: Taskhistogramm der Volume-Rendering-Applikation auf der IBM-Maschine (Szene „head").

Tasktyps z. B. durch Zugriffskonflikte hinweist.

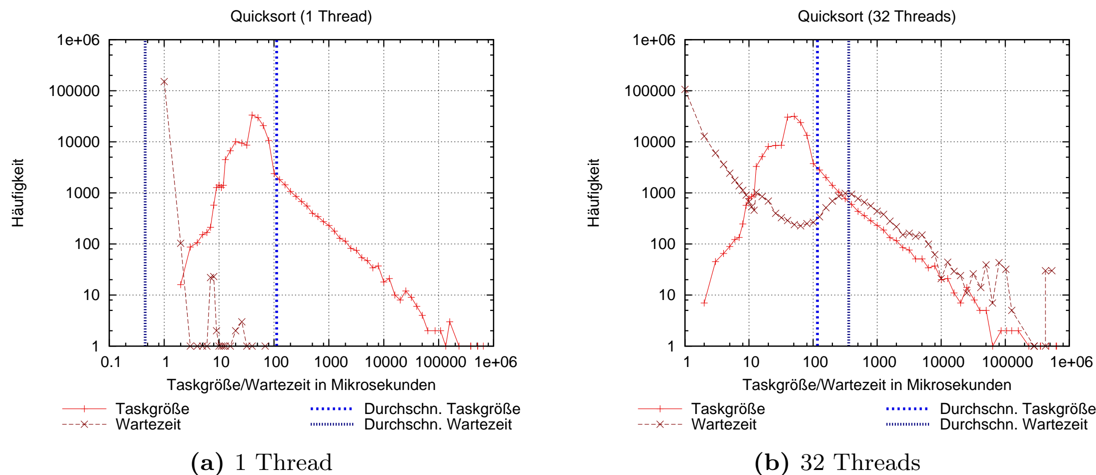
Die Wartezeiten steigen bei 32 Threads stark an, da der Overhead des Taskpools einen großen Einfluss hat. Die durchschnittliche Wartezeit ist mit $7,8\,\mu s$ fast zehn mal größer als bei der Ausführung mit zwei Threads. Dies ist schon mehr als 25 Prozent der durchschnittlichen Taskgröße. Zudem zeigt das Histogramm, dass gerade im Bereich längerer Wartezeiten die Vorkommen im Graph ähnlich häufig sind wie die entsprechenden Taskgrößen.

Die ansteigende Taskgröße weist auf ein Problem innerhalb der Applikation hin, während die Wartezeiten sowohl auf Probleme innerhalb der Taskverwaltung als auch auf zu geringe Taskerzeugung innerhalb der Applikation hinweist.

7.3.4 Quicksort

Die Quicksort-Applikation ist eine interessante Applikation, da die Taskstruktur besondere Anforderungen an den Taskpool stellt. Die relativ gute Skalierbarkeit wird durch die Taskhistogramme bestätigt. Bei sequenzieller Ausführung (Abbildung 7.8a) ist zu erkennen, dass es nur wenige sehr kleine Tasks gibt. Dies liegt an der Taskerzeugung, die ab einer bestimmten Feldgröße keine neuen Untertasks mehr erzeugt. Durch die im Idealfall halbierte Feldgröße für die jeweils beiden Untertasks steigt die Anzahl der Tasks kleinerer Größe exponentiell an, was sehr gut am geradlinigen Verlauf in der logarithmischen Darstellung zu sehen ist. Wichtiger ist jedoch, dass sich die Taskgröße bei 32 Threads (Abbildung 7.8b) nur unwesentlich von $111,8\,\mu s$ bei einem Thread auf $116,9\,\mu s$ erhöht hat. Der Graph der Taskgrößen ist nahezu identisch. Da die Anzahl ausführender Threads keinen Einfluss auf den Rechenaufwand eines Tasks hat, ist dieses Ergebnis zu erwarten. Allerdings zeigen die Messungen auch, dass es kaum Zugriffskonflikte gibt und auch die Speicherzugriffe keine Limitierung darstellen. Daher ist zu erwarten, dass auch diese Applikation mit einer größeren Anzahl von Prozessoren immer noch gute Ergebnisse erzielt.

Allerdings ist bekannt, dass diese Applikation nicht linear skalieren kann, da anfangs



(a) 1 Thread

(b) 32 Threads

Abbildung 7.8: Taskhistogramm der Quicksort-Applikation auf der IBM-Maschine (Feldgröße 100.000.000).

nicht genug Tasks für alle Threads vorhanden sind (vgl. Abschnitt 3.5). Dies ist in den Histogrammen in der Wartezeit zu sehen. Sie steigt von durchschnittlich $0,5\,\mu s$ bei sequenzieller Abarbeitung auf $360\,\mu s$ bei 32 Threads an. Zudem gibt es sehr viele große Wartezeiten und generell sind die Vorkommen der Wartezeiten in weiten Teilen im Graph über den Vorkommen der Tasklaufzeiten. Dabei wird mehrfach über ein halbe Sekunde auf neue Tasks gewartet. Die Wartezeiten weisen also direkt darauf hin, dass nicht genug Tasks zur Ausführung vorhanden sind, da bei 32 Threads mehr Zeit mit Warten verbraucht wird als mit der eigentlichen Taskausführung. Dies müsste innerhalb der Applikation gelöst werden, was aber bei der vorliegenden Implementierung der Quicksort-Applikation nicht ohne Weiteres möglich ist.

7.4 Skalierbarkeitsvorhersage

Die erfassten Profiling-Informationen können nicht nur zur Bewertung der Tasks einer Applikation verwendet werden. Die Informationen aus Experimenten für eine bestimmte Ausführungsplattform mit einer geringen Anzahl von verwendeten Prozessoren kann für eine Laufzeitvorhersage für eine größere Anzahl von Prozessoren benutzt werden. In diesem Abschnitt soll daher betrachtet werden, wie die Informationen über Tasklaufzeiten und Wartezeiten zur Berechnung eines zu erwartenden Speedups verwendet werden können. Ziel soll dabei die Bestimmung einer oberen Schranke für den Speedup sein.

Da Tasks aus Sicht des Taskpools unabhängig voneinander sind, ist die Summe der Tasklaufzeiten dividiert durch die Anzahl der teilnehmenden Prozessoren die zu erwartende Laufzeit im Idealfall. Trotzdem können Tasks andere Tasks beeinflussen, zum Beispiel wenn Locks für Datenstrukturen benötigt werden, ein Task die Speicherbandbreite ausnutzt und andere Tasks dadurch ausbremst oder durch schreibenden Zugriff auf gemeinsame Daten Cache-Fehlzugriffe häufiger auftreten. Da dies applikationsabhängig ist und nicht ohne Wei-

teres vom Taskpool erkannt werden kann, wird die Veränderung der Tasklaufzeiten für Messungen mit einer verschiedenen Anzahl von Prozessoren benutzt, um auf die vermutete Tasklaufzeit für eine bestimmte Anzahl von Prozessoren zu schließen.

Die zweite wichtige Information ist die Wartezeit, also die Zeit, die nicht in der Applikation verbracht wird, sondern im Taskpool benötigt wird, um neue ausführungsbereite Tasks zu finden bzw. auf solche zu warten. Auch hier wird die Veränderung der Wartezeit mit steigender Anzahl von Prozessoren benutzt, um die tatsächliche Wartezeit für eine größere Anzahl von Prozessoren vorherzusagen. Da alle Prozessoren einzeln die eigenen Wartezeiten messen und entsprechend überlappende Wartezeiten auch mehrfach erfasst werden, bestimmt die Summe der Wartezeiten dividiert durch die Anzahl der Prozessoren ebenfalls die zu erwartende Wartezeit, wenn keine weiteren Einflüsse auftreten.

Das Ziel der Laufzeitvorhersage ist es, für jeden Tasktyp einzeln unabhängig von den anderen Tasktypen eine zu erwartende durchschnittliche Taskgröße und Wartezeit zu bestimmen. Mit der Summe beider Werte über alle Tasktypen lässt sich dann daraus eine Abschätzung für die Ausführungszeit bestimmen, deren Berechnung für eine untere Schranke der Laufzeit (und somit obere Schranke des Speedups) von einer optimalen parallelen Ausführung ausgehen kann. Treten zusätzliche Störungen auf, ist die tatsächliche Laufzeit entsprechend noch geringer.

7.4.1 Methoden der Extrapolation

Ziel der Laufzeitvorhersage soll die Bestimmung eines Speedups für eine beliebige Anzahl von Prozessoren sein, ausgehend von mehreren Profiling-Messungen für eine jeweils unterschiedliche, jedoch kleinere Anzahl verwendeter Prozessoren. Zunächst soll betrachtet werden, welche Methode zur Berechnung der zu erwartenden Taskgrößen und Wartezeiten geeignet ist. Aufgrund der irregulären Natur der Applikationen kann eine genaue Vorhersage kaum erfolgen. Dennoch soll es möglich sein, ungefähre Aussagen über die zu erwartende Performance treffen zu können, mit denen insbesondere zu erwartende Performance-Einbrüche vorhergesagt werden können.

Grundlage der Berechnungen sind folgende Faktoren, die jeweils mit Hilfe des Profilings für jeden Tasktyp T einzeln und für eine bestimmte Anzahl benutzter Prozessoren p bestimmt wurden:

- durchschnittliche Taskgröße $ts_avg_{T,p}$
- Anzahl der ausgeführten Tasks $nr_{T,p}$
- durchschnittliche Wartezeit $wt_avg_{T,p}$

Die Vorhersage soll eine zu erwartende Taskgröße, Wartezeit und die Anzahl der ausgeführten Tasks für größere Prozessorzahlen berechnen, sodass daraus der Speedup abgeschätzt werden kann. Zwar ist die Anzahl der ausgeführten Tasks in den betrachteten Applikationen konstant, dennoch soll hier dieser Wert mit in die Berechnungen einfließen, um auch Applikationen betrachten zu können, bei denen dies nicht so ist. Für die Hochrechnung auf eine gegebene Prozessoranzahl pp (*predicted_p*) werden vorerst folgende Berechnungsschemata betrachtet:

1. Extrapolation nach dem *Schema von Neville*.

2. Paarweise lineare Extrapolation für alle Paare $(p1, p2)$ mit $p1 < p2$, wobei jeweils für $p1$ und $p2$ Messwerte für $(ts_avg_{T,p1}, nr_{T,p1}, wt_avg_{T,p1})$ bzw. $(ts_avg_{T,p2}, nr_{T,p2}, wt_avg_{T,p2})$ zur Verfügung stehen. Verwendung des maximalen extrapolierten Wertes.

Beide Ansätze werden im Folgenden kurz beschrieben.

Extrapolation mit *Neville*-artigen Algorithmen

Üblicherweise werden für die Extrapolation komplexe Verfahren betrachtet, um nicht lineares Verhalten modellieren zu können. Gegeben sei dabei eine Menge von Stützpunkten (x_i, y_i) mit denen ein y an einer Stelle x bestimmt werden soll, wobei $0 \leq i \leq n$. Neville-artige Algorithmen berechnen schrittweise aus einer Menge von bisher betrachteten Werten (polynomielle) Funktionen höherer Ordnung, um schließlich den eigentlichen Wert y an der Stelle x bestimmen zu können. Für eine genaue Beschreibung sei z. B. auf [123, S.72-78] verwiesen. Zur Extrapolation haben sich rationale Funktionen als besser geeignet herausgestellt als polynomielle Funktionen, da insbesondere das Verhalten außerhalb der Stützstellen weniger starken Schwankungen unterworfen ist.

Die Extrapolation erfolgt auf Basis folgender Vorschrift für einen gesuchten Wert bei gegebenem x:

$$\begin{aligned} T_{i,0} &= y_i \\ T_{i,-1} &= 0 \\ T_{i,k} &= T_{i,k-1} + \frac{T_{i,k-1} - T_{i-1,k-1}}{\frac{x-x_{i-k}}{x-x_i}\left[1 - \frac{T_{i,k-1}-T_{i-1,k-1}}{T_{i,k-1}-T_{i-1,k-2}}\right] - 1} \end{aligned}$$

mit $1 \leq k \leq i$. Der gesuchte Wert ist dann $T_{n,n}$.

Experimente mit gemessenen Profiling-Informationen zeigten jedoch, dass dieses Verfahren weniger geeignet ist. Problematisch sind erstens mögliche Polstellen insbesondere außerhalb der größten Stützstelle. Außerdem kann aufgrund der verwendeten Funktionen unter Umständen ein Umkehren der Entwicklungstendenz auftreten, d. h. obwohl alle Stützstellen ein Ansteigen erwarten lassen, kann außerhalb der Stützstellen wieder ein Abfallen berechnet werden (zudem hier noch relativ weit entfernt, z. B. 32 Threads gegenüber letzter Stelle mit 16 Threads). Dies entspricht jedoch nicht dem Erwartungswert (und tatsächlich auch nicht dem bei 32 Threads gemessenen Wert). Abbildung 7.9 zeigt die Ergebnisse für eine Extrapolation auf Basis polynomieller und rationaler Funktionen am Beispiel von beispielhaft ausgewählten Tasklaufzeiten. Da die Tasklaufzeiten im Beispiel mit größerer Prozessorenzahl ansteigen, ist zu erwarten, dass mit noch größerer Prozessorenzahl eine mindestens gleich große Laufzeit ermittelt wird. Beide Funktionsklassen sind in dem Beispiel zu weit weg vom Erwartungsbereich. Deshalb wird im Folgenden ein Verfahren beschrieben, das auf die speziellen Gegebenheiten des Task-Profilings abgestimmt ist.

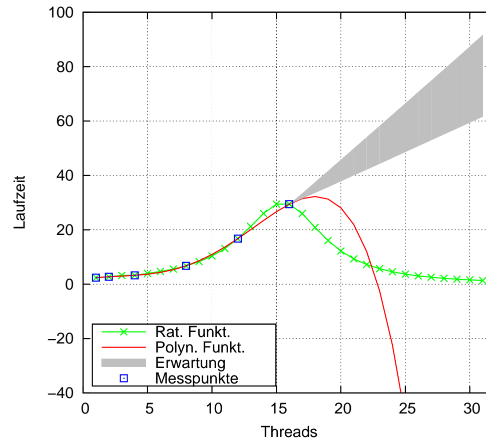


Abbildung 7.9: Beispielextrapolation für gegebene Messwerte.

Extrapolation einzelner Paare

Allgemein lässt sich für zwei Punkte (x_1, y_1) und (x_2, y_2) eine lineare Extrapolation für y_3 bei einem gegebenen Punkt x_3 wie folgt berechnen:

$$y_3 = y + m \cdot dx.$$

Der Anstieg m lässt sich aus den beiden gegebenen Punkten (x_1, y_1) und (x_2, y_2) berechnen:

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

und der Abstand dx kann bezüglich des Basispunkts x_2 berechnet werden:

$$dx = x_3 - x_2.$$

Somit kann der gesuchte Wert y_3 berechnet werden mit:

$$y_3 = y_2 + \frac{y_2 - y_1}{x_2 - x_1} (x_3 - x_2).$$

Entsprechend kann mit dieser Formel für ein gegebenes Paar von Messungen der Taskgröße bzw. Wartezeit für zwei unterschiedliche Werte von benutzten Threads ein Extrapolationswert berechnet werden. Gegeben sei für einen spezifischen Task T zwei Messpunkte $p1 \in N$ und $p2 \in N$ mit $p1 < p2$. Dann berechnen sich die extrapolierten Werte $v \in \{ts_avg, nr, wt_avg\}$ für $pp \in N > p2$:

$$v_{T,pp} = v_{T,p2} + \frac{(v_{T,p2} - v_{T,p1})}{p2 - p1} (pp - p2)$$

bzw. üblicherweise geschrieben als

$$v_{T,pp} = v_{T,p2} + \frac{pp - p2}{p2 - p1} (v_{T,p2} - v_{T,p1}). \tag{7.2}$$

Für alle Messungen mit einer bestimmten Anzahl benutzter Threads kann daraus jeweils paarweise ein Extrapolationswert für die Taskgröße, die Wartezeit und die Anzahl ausgeführter Tasks berechnet werden. Der größte Wert kann dann als Abschätzung und die Berechnung des Speedups benutzt werden.

Gewichtete Extrapolation basierend auf paarweise extrapolierte Werte

Bei der Benutzung des maximalen oder durchschnittlichen Anstiegs der paarweise extrapolierten Paare wird die Tendenz im Änderungsverhalten des Anstiegs außen vorgelassen. Die Polynomextrapolation kann dies mit berücksichtigen, liefert aber zu ungenaue Ergebnisse. Deshalb werden folgende Alternativen betrachtet:

1. Verwendung einer vorgegebenen Menge von Kurven, sodass mittels „Kurvenfitting“ Lösungen für den Zielwert bestimmt werden. Die Funktion mit dem kleinsten Fehler bei den Messstellen wird benutzt, um einen Wert an gesuchter Stelle zu berechnen.
2. Das Wertepaar mit dem geringsten Abstand zum Zielwert wird als Basis verwendet; andere Wertepaare werden aber tendenziell mit berücksichtigt.

Alternative 1: Untersuchungen mit gemessenen Werten zeigten, dass beispielsweise die Benutzung einer linearen Funktion zwar einen viel höheren Fehler für die Werte 1-16 Threads hatte als eine quadratische Funktion, allerdings für die Berechnung von 32 Threads einen viel besseren Wert lieferte. Es müsste also eine geeignete Menge von Funktionen gefunden werden, wobei dennoch nicht aus guten Fitting-Parametern auf eine gute Vorhersage geschlossen werden kann.

Alternative 2: Basis der Berechnungen soll das Wertepaar sein, das den geringsten Abstand zum gesuchten Wert hat. Dies kann z. B. bei einer typischen Messung der Profiling-Parameter mit 1, 2, 4, 8 und 16 Threads das Paar $(8, 16)$ sein, wenn man eine Vorhersage für die Benutzung von 32 Threads treffen möchte. Deshalb soll dieses Wertepaar benutzt werden, um damit einen linear extrapolierten Wert zu berechnen. Dann sollen die anderen Wertepaare benutzt werden, um diesen Wert zu korrigieren.

Dazu wird zuerst ein Gewichtungsfaktor eingeführt, der die Relevanz eines extrapolierten Wertes beschreibt. Dabei soll ein bestimmtes Paar als Referenzpaar den Faktor $1,0$ bekommen, welches die genauesten Informationen bereitstellt. Alle weiteren Paare sollen entsprechend ihrer Genauigkeit einen kleineren Wert bekommen.

Gesucht ist also eine Funktion, die für eine Menge von Paaren über $N \times N$ ein Gewicht g berechnet. Sei $i_1, i_2, ..., i_n$ eine echt aufsteigende Reihe von Messpunkten, so soll das Tupel (i_{n-1}, i_n) das größte Gewicht $g \leq 1$ bekommen, alle anderen einen entsprechend kleineren Wert. Das Tupel $(1, 2)$ soll das kleinste Gewicht $0 < g$ erhalten.

Zur Herleitung der Funktion sind folgende Bedingungen zu berücksichtigen:

- Der Abstand zum Zielwert (oder größtem Referenzwert) muss betrachtet werden, da Aussagen basierend auf einer Messung mit einer kleineren Anzahl von Threads weniger genaue Hochrechnungen zulassen als Aussagen basierend auf einer höheren Anzahl von Threads.

- Wie sollen zwei sich überlappende Paare geeignet gewichtet werden (z. B. $(8, 11)$ und $(9, 10)$)?
- Soll nur die absolute Entfernung der Messwerte vom Zielwert berücksichtigt werden oder auch die relative Distanz? Hat beispielsweise das Paar $(8, 16)$ ein höheres Gewicht als $(12, 16)$, da bei einem Zielwert von 32 nur die dreifache Entfernung zu extrapolieren ist?

Es sei im Folgenden definiert: $p1$ und $p2$ ist jeweils ein Messpunkt und pp der Zielwert zur Extrapolation mit $p1 < p2 < pp$. Dann kann man drei verschiedene Strecken definieren:

$$d1 = p2 - p1 \tag{7.3}$$

$$d2 = pp - p2 \tag{7.4}$$

$$d3 = pp - p1 \tag{7.5}$$

Die Strecke $d1$ ist also der Abstand der beiden Messpunkte, die Strecke $d2$ der Abstand vom größeren Messpunkt zum Zielpunkt und die Strecke $d3$ der Abstand vom kleineren Messpunkt zum Zielpunkt. Für die Betrachtungen der Extrapolation von Taskgrößen und Wartezeiten sind folgende Anforderungen zu betrachten:

1. $\frac{d2}{d1} \rightarrow \min$

   d. h. das Verhältnis der Strecken $p1, p2$ und $p2, pp$ sollte minimal sein, damit möglichst wenig Extrapolationsfehler auftreten,

2. $d2 \rightarrow \min$

   d. h. der größte Messpunkt sollte möglichst nah am Zielpunkt liegen, da dieser Messpunkt potenziell die genauesten Informationen liefert,

3. $d3 \rightarrow \min$

   d. h. auch der kleinste Messpunkt sollte nicht zu weit weg vom Zielpunkt liegen.

Diese Anforderungen sind nicht konfliktfrei, aus Punkt 2 folgt für Punkt 1 $d1 \rightarrow \max$. Allerdings ist $d3 = d1 + d2$, somit müsste für $d3 \rightarrow \min$ $d1$ minimal sein. Es existiert also ein Konflikt bei der Strecke $d1$:

1. $d1$ ist minimal, d. h. die beiden Messpunkte liegen möglichst nahe am Zielwert der Extrapolation. Dann ist $d3$ minimal, aber $\frac{d2}{d1}$ nicht minimal.
2. $d1$ ist maximal, d. h. die relative Strecke zum Zielwert ist am kleinsten. Damit ist $\frac{d2}{d1}$ minimal, aber $d3$ maximal.

In der praktischen Anwendung kann die zusätzliche Bedingung benutzt werden, dass Messpunkte mit konstanten oder wachsenden Abständen zu benutzen sind (z. B. die Folge $1, 2, 3, 4, 5, .., 16$ oder $1, 2, 4, 8, 16$). Zur Vereinfachung soll daher nur die Folge von Paaren

Messung für Threadpaar	Gewicht bei $pp = 32$
$(1, 2)$	30,0
$(2, 3)$	29,0
$(3, 4)$	28,0
$(4, 5)$	27,0
$(5, 6)$	26,0
$(6, 7)$	25,0
$(7, 8)$	24,0
$(1, 2)$	30,0
$(2, 4)$	14,0
$(4, 8)$	6,0

Tabelle 7.1: Beispiele der Extrapolationsgewichte $\frac{d2}{d1}$ für ausgesuchte Messpunkte.

$$(p_1, p_2), \ldots, (p_i, p_{i+1}), \ldots, (p_{n-1}, p_n)$$

mit $p_i < p_{i+1}$ betrachtet werden, wobei zusätzlich gelten soll:

$$(p_{i+1} - p_i) \leq (p_{i+2} - p_{i+1}).$$

Mit anderen Worten werden aufsteigende Messpunkte mit jeweils mindestens gleich großem Abstand benutzt. Für solche Paare kann gezeigt werden, dass der Quotient $\frac{d2}{d1}$ eine monoton fallende Reihe ist, bzw. das Reziproke als Gewicht verwendet werden kann. Dazu betrachtet man zwei benachbarte Paare, also $p' = (p_i, p_{i+1})$ und $p'' = (p_{i+1}, p_{i+2})$ für ein beliebiges i. Analog zu den Gleichungen (7.3)-(7.5) sei $d1', d2', d3'$ und $d1'', d2'', d3''$ definiert. Für diese Werte gilt:

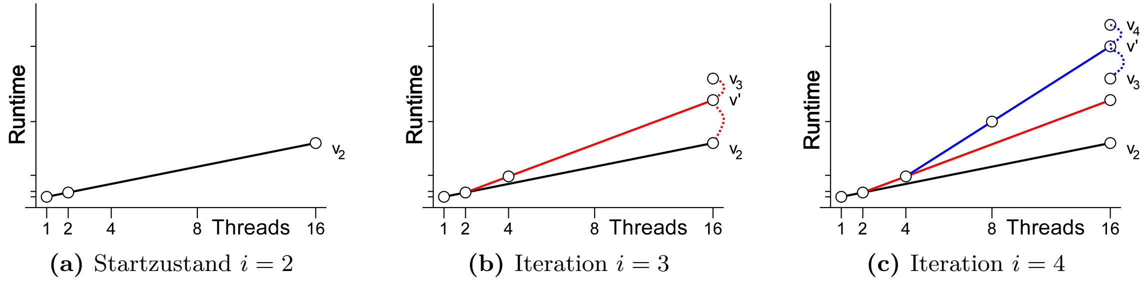
$$\begin{aligned} d1' = (p_{i+1} - p_i) &\leq d1'' = (p_{i+2} - p_{i+1}) \\ d2' = (pp - p_{i+1}) &> d2'' = (pp - p_{i+2}) \\ d3' = (pp - p_i) &> d3'' = (pp - p_{i+1}) \end{aligned} \tag{7.6}$$

Aus $d3'' = d2'$ folgt

$$\begin{aligned} d1'' + d2'' &= d2' \\ d2'' &= d2' - d1'' \\ \frac{d2''}{d1''} &= \frac{d2' - d1''}{d1''} = \frac{d2'}{d1''} - 1 \end{aligned}$$

Wegen Gleichung (7.6) gilt mit $d1' \leq d1''$ deshalb

$$\frac{d2''}{d1''} = \frac{d2'}{d1''} - 1 \leq \frac{d2'}{d1'} - 1 < \frac{d2'}{d1'}$$



(a) Startzustand $i = 2$ **(b)** Iteration $i = 3$ **(c)** Iteration $i = 4$

Abbildung 7.10: Exemplarische Extrapolation für $pp = 16$ bei Messungen für 1, 2, 4 und 8 Threads.

Das Verhältnis der Strecke zum Zielwert der Extrapolation und des Abstandes der beiden Messpunkte beschreibt also für jeweils benachbarte Paare ein echt monoton fallendes Gewicht, dass sich für die gewichtete Extrapolation eignet. Tabelle 7.1 zeigt exemplarisch die Gewichte für zwei verschiedene Messreihen. Da Messpunkte mit kleinerem Quotienten $\frac{d2}{d1}$ das größere Gewicht bei der Extrapolation bekommen sollen, wird das Reziproke benutzt.

Die eigentliche Extrapolation verläuft nun iterativ. Für das erste Paar (p_1, p_2) wird entsprechend Gleichung (7.2) ein Startwert v_2 für den gesuchten Punkt pp linear extrapoliert. Beginnend bei $i = 3$ wird nun für jedes aufsteigende i eine neuer Wert v_i wie folgt berechnet:

1. Lineare Extrapolation eines Zwischenwertes v' basierend auf den Messwerten für p_{i-1} und p_i entsprechend Gleichung (7.2).

2. Bestimmung des Gewichts $g_a = \frac{pp - p_{i-1}}{p_{i-1} - p_{i-2}}$ und $g_b = \frac{pp - p_i}{p_i - p_{i-1}}$.

3. Bestimmung eines neuen v_i:

   Dabei soll berücksichtigt werden, dass ein kleinerer vorheriger Wert v_{i-1} bei einem größeren Zwischenwert v' auf ein größeres $v_i > v'$ schließen lässt (Berücksichtigung der Tendenz). Ohne Gewichtung könnte dies mit

$$v'' = v' + (v' - v_{i-1})$$

   berechnet werden. Um die Messpunkte entsprechend zu gewichten, wird das Reziproke der zuvor berechneten Gewichte g_a und g_b für den Punkt v'' und v' wie folgt berücksichtigt:

$$v_i = \frac{\frac{1}{g_a}(v' + (v' - v_{i-1})) + \frac{1}{g_b}v'}{\frac{1}{g_a} + \frac{1}{g_b}}$$

Der finale Extrapolationswert ist dann das v_i berechnet für den letzten Messpunkt i mit der größten Anzahl von Threads.

Abbildung 7.10 illustriert die Extrapolation am Beispiel einer Messung für 1, 2, 4 und 8 Threads, um damit eine Vorhersage für die Laufzeit mit 16 Threads zu berechnen. Zur

Initialisierung (Abbildung 7.10a) wird der Wert v_2 als lineare Extrapolation der gemessenen Werte für einen und zwei Threads berechnet. In der ersten Iteration für $i = 3$ (Abbildung 7.10b) wird der Zwischenwert v' für die Messungen mit zwei und vier Threads berechnet. Aus diesem Wert und dem vorherigen Wert v_2 wird entsprechend der Gewichte der beiden Messungen ein neuer Wert v_3 berechnet, der im Beispiel oberhalb des Zwischenwertes v' liegt, da der vorherige Wert v_2 einen überlinearen Anstieg vermuten lässt. Der Einfluss des Wertes v_2 ist entsprechend des verwendeten Gewichtes geringer, sodass der neue Wert v_3 näher an v' liegt. Im letzten Iterationsschritt für $i = 4$ (Abbildung 7.10c) wird ausgehend von der Messung für vier und acht Threads wieder ein Zwischenwert v' berechnet und entsprechend der Gewichte für die Paare $(2, 4)$ und $(4, 8)$ wird aus v_3 und v' ein finaler Extrapolationswert v_4 berechnet.

7.4.2 Extrapolation am Beispiel der Radiosity-Applikation

Die Task-Analyse der Radiosity-Applikation brachte die unterschiedlichen Taskstrukturen der Applikation zum Vorschein. Mit der zuvor beschriebenen Extrapolationsstrategie kann für jeden Tasktyp isoliert eine Vorhersage berechnet werden, wiederum jeweils einzeln für die zu erwartende Taskgröße und Wartezeit. Die kombinierten Laufzeiten werden zu einer erwarteten Gesamtlaufzeit kombiniert und erlauben so die Berechnung eines Speedups. Da die gesammelten Informationen nur aus der Zwischenschicht des Frameworks stammen, fließen keine Details über die Applikation und die Taskpool-Implementierung in diese Extrapolation ein. Daher kann die Vorhersage natürlich keine exakten Werte liefern, sie soll aber möglichst ein oberes Limit für den zu erwartenden Speedup darstellen.

Für die hierarchische Radiosity-Applikation wurden die durchschnittlichen Taskgrößen und Wartezeiten jeweils für eine unterschiedliche Anzahl benutzter Threads protokolliert. Als Basis der Extrapolation werden die gemessenen Taskgrößen und Wartezeiten der vier verschiedenen Tasktypen jeweils für 1, 2, 4, 8 und 16 Threads verwendet.

Mit dem zuvor dargestellten Verfahren sind jeweils für 32 benutzte Threads aus diesen Messdaten die durchschnittliche Taskgröße avg_ts_T, die durchschnittliche Wartezeit avg_wt_T und die Anzahl ausgeführter Tasks nr_T für Tasktyp T berechnet worden. Entsprechend ergibt sich für die Gesamttaskausführungszeit

$$ts_T(p) = avg_ts_T(p) \cdot nr_T(p)$$

und die Gesamtwartezeit

$$wt_T(p) = avg_wt_T(p) \cdot nr_T(p)$$

jeweils für die vier Tasktypen $T \in \{RAY, REFINEMENT, AVERAGE, VISIBILITY\}$ und $p = 32$.

In Tabelle 7.2 sind die auf dieser Basis berechneten Daten aufgelistet. Zusätzlich ist die jeweilige tatsächliche Zeit in Prozent der berechneten Zeit dargestellt. Die meisten Zeiten wurden wie vorgesehen unterschätzt, am stärksten überschätzt wurde die Ausführungszeit für den „Refinement“-Task, der schon bei der Histogramm-Analyse als sehr kritischer Task identifiziert wurde. Insgesamt wurde die Ausführungszeit grob um ein Viertel unterschätzt, während die Wartezeit deutlich größer war als vorhergesagt, was aber auch durch die relativ

Radiosity Tasktyp	Berechnungszeit		Wartezeit	
	Berechnete Zeit in s	Gemessene Zeit (relativ)	Berechnete Zeit in s	Gemessene Zeit (relativ)
„RAY“	2,92	**198%**	0,13	**231%**
„REFINEMENT“	20,25	**60%**	0,30	**233%**
„AVERAGE“	1,75	**89%**	0,16	**75%**
„VISIBILITY“	22,14	**172%**	0,05	**720%**
Summe	47,06	**122%**	0,63	**234%**

Tabelle 7.2: Vorhergesagte und tatsächliche Ausführungszeit und Wartezeit bei 32 Threads für die Radiosity-Applikation (Szene „largeroom“, IBM-System).

kurzen Messzeiten erklärt werden kann. Das Extrapolationsverfahren wurde außerdem so ausgelegt, dass möglichst eine untere Schranke für die zu erwartende Laufzeit berechnet wird. Trotz der teilweise hohen Unterschiede zur realen Laufzeit wurde dies im Beispiel erreicht. Da keinerlei zusätzliche Informationen über die Applikation und die Taskpool-Implementierung außer der protokollierten Taskgrößen und Wartezeiten in die Vorhersageberechnung einflossen, ist das Ergebnis akzeptabel, insbesondere für die ohnehin nicht gut skalierende Applikation.

Mit diesen Werten lässt sich nun auch ein Speedup berechnen, indem die berechnete Ausführungszeit

$$t_{real,T}(p) = \frac{ts_T(p) + wt_T(p)}{p}$$

bzw. für alle Tasktypen

$$t_{real}(p) = \sum_T t_{real,T}(p)$$

in Relation zur gemessenen Zeit bei einem Thread gesetzt wird. Die vorhergesagte Laufzeit bei 32 Threads ist $\approx 1,49\,s$ bei tatsächlichen $\approx 1,85\,s$. Die sequenzielle Ausführung dauerte (ebenfalls bei Benutzung der Profiling-Komponente) $\approx 14,53\,s$, sodass der vorhergesagte Speedup

$$speedup_{est}(32) = 9,75$$

und der tatsächliche Speedup

$$speedup_{act}(32) = 7,87$$

beträgt. Anhand der Vorhersage kann also der Schluss gezogen werden, dass der Einsatz von 32 Threads auf dieser Maschine keine nennenswerten Laufzeitvorteile bietet, zudem der Speedup bei 16 Threads mit $9,5$ nur etwas unter dem vorhergesagten Speedup von $9,75$ bei 32 Threads liegt. Die absoluten Zeiten helfen auch, die wichtigsten Tasktypen zu identifizieren, die am meisten von einer Verbesserung profitieren können. Bei der Histogramm-Analyse wurden der „Average“-Task und der „Refinement“-Task als kritische Tasktypen

identifiziert. Die Werte hier geben die zusätzliche Information, dass der „Refinement“-Task einen wesentlich höheren Anteil an der tatsächlichen Laufzeit hat und somit die Gesamtlaufzeit deutlich mehr beeinflusst.

Zusammengefasst erlaubt die Histogramm-Analyse und die Speedup-Extrapolation die Identifizierung der kritischen Tasktypen einer taskbasierten Applikation, die am meisten von einer Verbesserung profitieren würden. Da die einzelnen Tasks isoliert voneinander betrachtet werden, muss die Extrapolation nicht verschiedene Trends im Anstieg der Taskgrößen oder Wartezeiten modellieren. Die Profiling-Komponente erlaubt eine Analyse ohne Änderungen an der Applikation oder der Taskpool-Implementierung. Zudem ist die Protokollierung mit relativ wenig Overhead verbunden und die Applikation kann während der Protokollierung parallel ausgeführt werden, sodass Engstellen aufgrund wechselseitigem Zugriffs aufgedeckt werden können.

7.5 Identifizierung von Cache-Fehlzugriffen aufgrund von False-Sharing

Nachdem die kritischen Tasks mit dem Profiling-Mechanismus identifiziert wurden, soll nun eine Methode betrachtet werden, mit der Probleme innerhalb von Tasks durch Cache-Fehlzugriffe erkannt werden können. Greifen mehrere Threads parallel auf gemeinsame Datenstrukturen zu, kann dies zu einer ungewollten Invalidierung des Caches der jeweils anderen Threads und somit zu einer erhöhten Ausführungszeit bzw. einer größeren Taskgranularität führen.

Einige Probleme können in den kritischen Tasks direkt erkannt werden, wenn z. B. eine häufige Verwendung von Locks die Skalierbarkeit limitiert. Wenn jedoch Datenstrukturen gemeinsam modifiziert werden, ist die Skalierbarkeit auch von den folgenden Faktoren abhängig:

- Skalierbarkeit der Datenstruktur:

  Hängt z. B. die Zugriffszeit von der Anzahl der teilnehmenden Threads ab? Beispielsweise steigt bei Verwendung einer zentralen Liste die Zugriffszeit bei größerer Anzahl von Threads an, wogegen bei verteilten Listen mit einer Liste pro Thread die Zugriffszeit konstant bleibt (wenn es zu keinen Kollisionen kommt).

- Nötige Synchronisationen zwischen den Threads:

  Skaliert die Synchronisationsmethode mit der Anzahl der Threads?

- Konstante Operationen:

  Die Zugriffe eines Threads auf seine lokalen Datenstrukturen sollten unabhängig von anderen Threads möglich sein (unbeachtet der nötigen Synchronisation). Insbesondere sollte der Programmlauf eines Threads keinen Einfluss auf den Programmlauf eines anderen Threads haben.

Während die ersten beiden Punkte Parameter der verwendeten Datenstruktur und deren Synchronisationsoperationen sind, beschreibt der dritte Punkt einen nicht trivialen Einfluss aufgrund der Architektur des Systems. Die Effizienz der eingesetzten Datenstrukturen

und Mechanismen zum Datenaustausch können von Hardware-Parametern abhängen. So spielt z. B. die Größe eines Caches eine Rolle und kann eine theoretisch effizient zugreifbare Datenstruktur behindern.

Auch im Kontext des Taskpools spielt False-Sharing eine große Rolle, wenn z. B. mehrere Threads auf verschiedene Adressen der gleichen Cache-Zeile zugreifen und so künstlich Cache-Fehlzugriffe mit entsprechend längeren Zugriffszeiten erzeugen. Dies ist deshalb relevant, da die Lastbalancierung einen Austausch von Tasks erfordert und so einzelne Taskstrukturen eines Threads im Laufe der Zeit in Listen anderer Threads abgelegt werden können. Es ist daher wichtig, die Datenstruktur so im Speicher abzulegen, dass dieser Effekt möglichst vermieden wird.

7.5.1 Messung der False-Sharing-Effekte

Um das Problem des False-Sharings zu identifizieren, können mehrere Indizien herangezogen werden. Die einfachste Methode mit dem geringsten Overhead ist die Beobachtung der User-Zeit bei Laufzeitexperimenten. Steigt diese Zeit mit einer höheren Anzahl von Threads an, obwohl die Datenstrukturen unabhängig von der Anzahl der Threads sind, deutet dies auf eine erhöhte Speicherzugriffszeit hin. Ursache dafür kann eine Limitierung der Busbandbreite oder eine erhöhte Anzahl von Cache-Fehlzugriffen sein. Allerdings gibt die Betrachtung der User-Zeit nur einen vagen Hinweis auf diese möglichen Probleme, insbesondere ist die genaue Identifizierung der verantwortlichen Programmstellen kaum möglich.

Des Weiteren kann mit Tools wie *papiex* [95], ein auf PAPI [33] aufbauendes Programm, eine Vielzahl an Hardware-Parametern protokolliert werden, die selbstständig von den Prozessoren während der Abarbeitung gezählt werden. Dazu gehört auch die Anzahl der Cache-Fehlzugriffe. Steigt diese für die gleiche Eingabe mit einer Erhöhung der Threadanzahl an, kann dies u. a. auf False-Sharing zurückzuführen sein.

Diese Messwerte stellen aber nur Hinweise dar, die keine Hilfe bieten, um die Problemstellen tatsächlich einzugrenzen. Deshalb gibt es aufbauend auf dem Analyse-Tool *valgrind* [116] Erweiterungen, um genauere Informationen zu erhalten. Im Speziellen kann das Plugin *cachetool* [135] benutzt werden, um recht einfach einen „Trace“ aller Speicheroperationen inklusive Speicherallokationen zu erzeugen. Mit Hilfe eines Analyse-Tools ist es dann möglich, False-Sharing innerhalb von Multi-Threaded-Applikationen zu lokalisieren.

7.5.2 Analyse-Tool „Valgrind“

Das Programm „valgrind“ [116] ist ein vielseitiges Analyse-Tool, um verschiedenartige Probleme in Applikationen zu finden. Das Tool arbeitet lediglich mit dem kompilierten Programm, bedarf also keinerlei Quellcodeinstrumentierung. Ein Programm kann damit also in dem Zustand analysiert werden, in dem es auch ausgeführt werden soll.

Die Vorgehensweise kann dabei wie folgt zusammengefasst werden. Das gegebene Programm wird mit allen benötigten Bibliotheken geladen. Die plattformabhängigen Assemblerinstruktionen werden in eine RISC-ähnliche Zwischendarstellung überführt und in Basisblöcken zusammengefasst. Diese Blöcke können dann von dem jeweiligen „Plugin“ instrumentiert werden, d. h. jede einzelne Instruktion kann ersetzt oder erweitert werden. So kann z. B. bei einem Speicherzugriff ein Aufruf einer eigenen Protokollierungsfunktion eingefügt

werden, die im Kontext des valgrind-Programms läuft, jedoch für das zu analysierende Programm nicht sichtbar ist. Dies erlaubt ein äußert flexibles Instrumentieren, allerdings geht dabei teilweise der Kontext der Ausführung verloren, da letztlich nur die Speicheradresse und die Assembleranweisung bekannt ist. Es ist jedoch möglich, über den Programmzähler und mit Debug-Informationen wieder die ursprüngliche Quellcodezeile zu ermitteln.

Die so instrumentierten Basisblöcke werden danach wieder in entsprechende Instruktionen der Zielplattform übersetzt und ausgeführt. Je nach Art der Instrumentierung kann dabei die Programmgeschwindigkeit erheblich verlangsamt sein, Faktoren von 10-100 sind durchaus realistisch.

Für die hier betrachtete Analyse der Speicherzugriffe ist es wichtig zu wissen, welche allokierten Speicherblöcke viele Cache-Fehlzugriffe aufweisen. Daher bietet „valgrind" eine weitere Schnittstelle für die Ersetzung von Standardbibliotheken an. Soll auf regulärem Weg protokolliert werden, wann ein Speicherblock allokiert wird, so müsste mittels Instrumentierung jeder Funktionsaufruf daraufhin überprüft werden, ob z. B. die Funktion *malloc* aufgerufen wird. Dies wäre eine erhebliche Schwierigkeit, da der Symbolname nicht zwangsläufig vorhanden ist und ohne diese Information die *malloc*-Funktion nicht von einer anderen unterschieden werden kann. Außerdem erschweren Compileroptimierungen wie z. B. *Inlining* dies weiter.

Daher können innerhalb eines „valgrind"-Plugins Funktionsaufrufe auf bestimmte Standardbibliotheken durch eigene Implementierungen ersetzt werden. Damit ist es möglich, bei einem *malloc*-Aufruf neben den eigentlichen Argumenten auch die genaue Programmposition und den ausführenden Thread zu identifizieren.

Ein wichtiges Kriterium zur Wahl von „valgrind" ist die Unterstützung von Multi-Threaded-Applikationen. Zwar führt „valgrind" das instrumentierte Programm nicht parallel aus, aber es führt die entsprechenden Threads zeitscheibenweise aus. Für die Lokalisierung von False-Sharing reicht das aber aus, wenn die Zeitscheibe kurz genug ist. Wurden in der Originalimplementierung noch 100.000 Basisblöcke ausgeführt, bevor ein anderer Thread an die Reihe kam, wurde dies in der für diese Arbeit modifizierten Variante auf 1000 bzw. 100 Basisblöcke reduziert. Der Overhead steigt zwar an, ist aber durch die Protokollierung jedes einzelnen Speicherzugriffes ohnehin so hoch, dass dieser zusätzliche Overhead nicht signifikant ist. Die Benutzung von verschieden großen Zeitscheiben (100 und 1000 Basisblöcke) erlaubt eine bessere Identifizierung, da so normale Fehlzugriffe z. B. bei Erstzugriffen von Fehlzugriffen aufgrund von False-Sharing quantitativ leichter zu unterscheiden sind. Die zu untersuchenden Fehlzugriffe existieren erst aufgrund der parallelen Abarbeitung, ein häufigeres Wechseln zwischen den Threads kann also die Anzahl dieser Fehlzugriffe erhöhen und sie somit von anderen Fehlzugriffen (z. B. Erstzugriff auf eine Speicheradresse) unterscheiden.

7.5.3 Cachetool-Plugin

Das Plugin „cachetool" der Cornell-Universität [135] protokolliert die durchgeführten Speicheroperationen eines Programmlaufs. Zu besseren Analyse wurde innerhalb dieser Arbeit das Plugin derart modifiziert, dass eine textbasierte Ausgabe erzeugt wird, die zwar eine größere Ausgabedatei erzeugt, aber eine Analyse mit Scriptsprachen wie *python* vereinfacht und fehlerresistenter macht. Da der Analyseschritt keine Komplettansicht der Trace-Daten

benötigt, muss die Ausgabe nicht vollständig gespeichert werden und kann durch Benutzung von *pipes* ohne Zwischenspeichern analysiert werden.

Darüber hinaus wurde die Bestimmung der aktuellen Programmposition und deren korrespondierenden Position im Quelltext korrigiert und erweitert, sodass auch Informationen über jeden einzelnen Thread gesammelt werden können.

In der bestehenden Implementierung wurden die wichtigsten Speicherallokierungsfunktionen entsprechend erweitert, um im Analyseschritt eventuelle Speicherzugriffskonflikte auf allokierte Speicherbereiche zurückzuführen.

7.5.4 Analysetool

Das in dieser Arbeit entwickelte Analysetool „ct-txt-analyser" simuliert auf Basis der Trace-Daten des modifizierten Cachetool-Plugins einen gemeinsamen Cache mit unbegrenzter Kapazität für alle Threads. Für jede zugegriffene Adresse wird eine Liste aller zugegriffenen Speicheradressen gespeichert, die jeweils die zugegriffenen Threads enthält. Ein Lesezugriff fügt den ausführenden Thread zu der Liste hinzu und protokolliert einen Lesefehlzugriff, falls er noch nicht in der Liste stand. Ein Schreibzugriff entfernt alle anderen Threads aus der Liste, sodass nur der ausführende Thread eingetragen ist. Gegebenenfalls wird ein Schreibfehlzugriff protokolliert, falls der Thread vorher nicht in der Liste enthalten war oder zwischenzeitlich aufgrund von Zugriffen anderer Threads wieder entfernt wurde. Entsprechend wird bei wechselseitigem Zugriff auf die gleiche Adresse der jeweils andere Thread entfernt, sodass jeder Zugriff als Cache-Fehlzugriff protokolliert werden kann.

Da für die Analyse von False-Sharing Fehlzugriffe auf gleiche Adressen nicht relevant sind, sondern Zugriffe auf unterschiedliche Adressen einer einzelnen Cache-Zeile, werden die jeweiligen Adressen auf die Anfangsadressen der entsprechenden Cache-Zeilen reduziert. Als Programmoption kann die Größe der Cache-Zeile vorgegeben werden, sodass die gleiche Trace-Datei für die Simulation unterschiedlicher Caches benutzt werden kann.

Für jeden Fehlzugriff wird zusätzlich die genaue Programmstelle protokolliert. Wird das ausgeführte Binärprogramm mit angegeben und enthält entsprechende Debug-Informationen, wird direkt die zu einem Fehlzugriff gehörende Programmzeile protokolliert. Die Ausgabe erfolgt sortiert nach der Häufigkeit der Fehlzugriffe für jede Cache-Zeile wie in Abbildung 7.11 beispielhaft dargestellt. Die hohen Vorkommen von Fehlzugriffen sind in den Funktionen zu finden, die entsprechend verändert werden müssen, um das Problem zu beheben (im Beispiel die Funktion „get_interaction"). Anhand der Zeilennummer (die im gekürzten Beispiel nicht abgebildet ist) kann so im Quellcode nachvollzogen werden, welche Datenstruktur in welchem Kontext Fehlzugriffe aufweist, sodass natürlich applikationsabhängig eine Veränderung vom Programmierer durchgeführt werden kann.

7.6 Verbesserung der Radiosity-Applikation

Die vorgestellten Profiling-Mechanismen sind geeignet, die problematischen Tasktypen einer Applikation zu identifizieren, um Verbesserungen durchführen zu können.

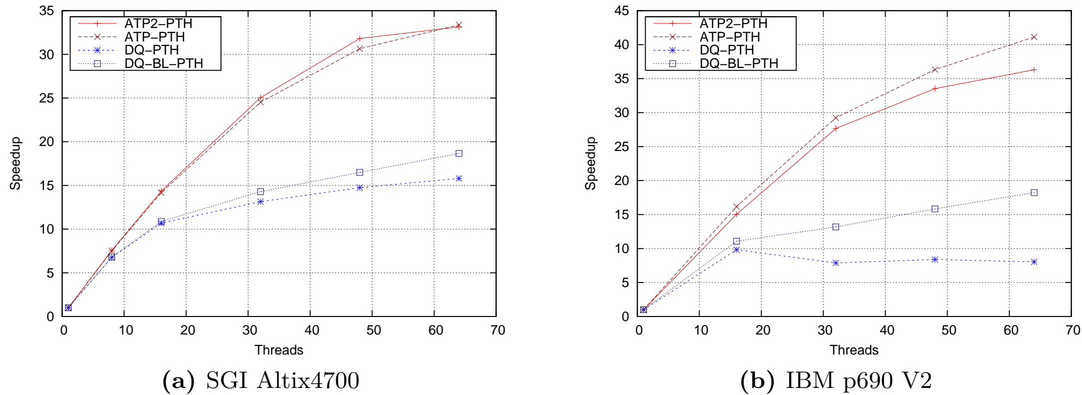
Bei der Histogramm-Analyse der Radiosity-Applikation wurden die „Refinement"-Tasks und die „Average"-Tasks als schlecht skalierend identifiziert. Bei der Vorhersage wurde dem „Refinement"-Task einer der größten Anteile an der Laufzeit vorhergesagt. Deshalb

```
Here's a list of memory blocks with cache misses:
  664cd40 (1194 misses)
    in block 55d7030 (207772512 bytes) allocated in 0x402BE6:
    Accesses in:
      1038 x: 0x40A87B: get_interaction
      136 x: 0x40A747: free_interaction
        8 x: 0x40A884: get_interaction
        6 x: 0x40A8B3: get_interaction
        5 x: 0x40A750: free_interaction
        1 x: 0x40D7C2: tplock_init
  664cd80 (1181 misses)
    in block 55d7030 (207772512 bytes) allocated in 0x402BE6:
    Accesses in:
      1035 x: 0x40A896: get_interaction
      137 x: 0x40A762: free_interaction
        4 x: 0x40A606: bf_error_analysis
        1 x: 0x406B75: create_visibility_tasks
        1 x: 0x405F41: get_elem_stat
        1 x: 0x40430A: compute_visibility_values
        1 x: 0x40A808: init_interactionlist
        1 x: 0x40434F: compute_visibility_values
  5843140 (405 misses)
    in block 55d7030 (207772512 bytes) allocated in 0x402BE6:
    Accesses in:
      381 x: 0x40A90B: get_element
       10 x: 0x40B442: process_rays3
        6 x: 0x40BF6B: process_rays
        6 x: 0x40B5B7: subdivide_element
        1 x: 0x40A710: insert_vis_undef_interaction
        1 x: 0x404CA7: init_patchlist
  ...
```

Abbildung 7.11: Auszug einer False-Sharing-Analyse der Radiosity-Applikation.

ist eine Verbesserung dieses Tasks sinnvoll. In Abbildung 7.11 wurde ein Ausschnitt der Ausgabe für die Analyse der Radiosity-Applikation mit einer Cache-Zeilengröße von 64 Byte dargestellt. Sortiert nach der Anzahl der Fehlzugriffe fallen hier die ersten beiden Bereiche besonders auf. Die wesentlichen Fehlzugriffe sind innerhalb des „Refinement“-Tasks in den Funktionen *get_interaction* und *free_interaction* auszumachen. Diese Funktionen werden bei der Unterteilung der Szene für die Allokation neuer Interaktionen zwischen unterteilten Elementen benutzt. Da Tasks im Laufe der Abarbeitung zwischen Threads ausgetauscht werden, werden diese Speicherelemente in Freilisten verschiedener Threads abgespeichert und wiederverwendet. Bei der Wiederverwendung invalidieren die Schreibzugriffe auf diese Struktur die gesamte Cache-Zeile in den Caches der anderen Prozessoren, sodass Zugriffe anderer Threads auf im Speicher benachbarte Blöcke zu erhöhten Fehlzugriffen führen.

Da sich diese Struktur innerhalb des „Refinement“-Tasks befindet, der als kritischer Tasktyp identifiziert wurde, ist eine Verbesserung erfolgversprechend. Daher wurde die Allokation und Freigabe der entsprechenden Speicherblöcke durch die Benutzung der Cache-optimierten Speicherliste „memlist“ ersetzt, die als Bestandteil des KOALA-Frameworks zur Verfügung steht (vgl. Kapitel 5.3.2). Abbildung 7.12 zeigt die Resultate der so verbesserten Radiosity-Applikation auf der SGI-Maschine und einer neuen Ausbaustufe der IBM-Maschine. Die Power4+-Prozessoren dieser Maschine wurden durch Power6-Prozessoren ersetzt, die mit $4,7$ GHz getaktet sind und mit SMT zwei Threads pro Prozessorkern unterstützen.



(a) SGI Altix4700 **(b)** IBM p690 V2

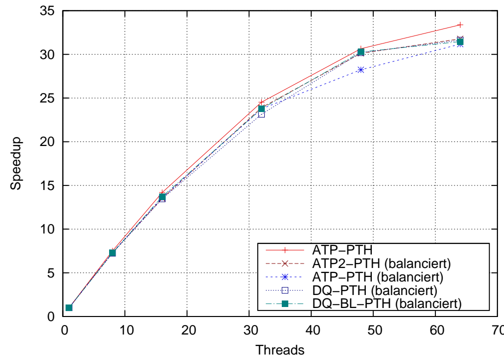
Abbildung 7.12: Speedups der verbesserten hierarchischen Radiosity-Applikation für die Szene „room11“.

Auf der SGI-Maschine gab es mit der Radiosity-Applikation die größten Probleme, mit den Änderungen (Abbildung 7.12a) ist die Skalierbarkeit aber wesentlich verbessert wurden. Bei 32 Threads ist Speedup der besten Taskpool-Implementierung ≈ 25 und bei 64 Threads wird immer noch ein Speedup von ≈ 33 erreicht. Besonders auffällig ist der große Abstand der adaptiven Taskpools zu den konventionellen Implementierungen mit verketteten Listen. Die blockorientierte Implementierung „DQ-BL“ erreicht einen Speedup von ≈ 18 und ist somit nur fast halb so schnell wie die adaptiven Implementierungen.

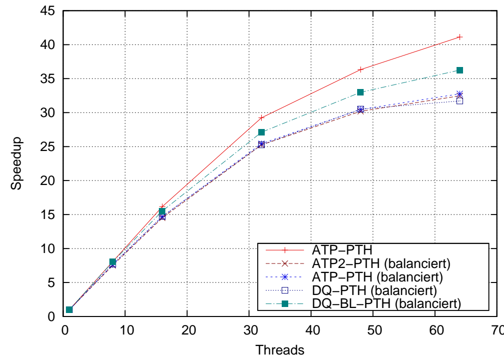
Auf der IBM-Maschine (Abbildung 7.12b) zeigt sich ein ähnliches Bild. Die adaptiven Taskpools erreichen bei der maximalen Prozessorzahl von 32 einen sehr guten Speedup von etwas unter 30, während der blockorientierte Taskpool weniger als die Hälfte erreicht. Die Nutzung der SMT-Threads ist auf dieser Architektur vorteilhaft, die beste Implementierung erreicht einen Speedup von mehr als 40.

Um den gravierenden Abstand der konventionellen Taskpool-Implementierungen zu den adaptiven Implementierungen zu untersuchen, wurde eine Modifikation im Taskpool-Frontend vorgenommen, von der also alle Backend-Implementierungen betroffen sind. Die initial erzeugten Tasks werden vor der eigentlichen parallelen Abarbeitung mit *tp_run* nochmals auf alle teilnehmenden Threads und somit deren Datenstrukturen balanciert. Da das Radiosity-Verfahren mehrere Iterationen und Phasen durchläuft, wird dies also mehrfach während der Gesamtabarbeitung durchgeführt. Ziel dieser einfachen Modifikation ist es, den Anfangszustand zu balancieren, um eventuell weniger Task-Stealing ausführen zu müssen. Die in Abbildung 7.13 gezeigten Resultate stellen die wesentlich bessere Performance der konventionellen Taskpools gegenüber der Variante ohne initiale Balancierung dar. Auf der SGI-Maschine (Abbildung 7.13a) liegen nun die Laufzeiten aller Implementierungen nah beieinander, die zusätzlich dargestellte beste Implementierung der vorherigen Variante ohne zusätzliche Balancierung ist aber deutlich schneller. Auf der IBM-Maschine (Abbildung 7.13b) zeigt sich ein ähnliches Bild, allerdings ist nun der blockorientierte Taskpool deutlich besser als die anderen Implementierungen inklusive der adaptiven Taskpools. Der adaptive Taskpool ohne zusätzliche Balancierung ist jedoch nochmals wesentlich schneller.

Es zeigt sich also, dass die Verbesserung innerhalb des kritischen „Refinement“-Tasks



(a) SGI Altix4700



(b) IBM p690 V2

Abbildung 7.13: Speedups der verbesserten hierarchischen Radiosity-Applikation mit zusätzlicher Balancierung während der Initialisierung (Szene „room11“).

tatsächlich zu einer Verbesserung der Skalierbarkeit führt. Die Untersuchung hat zudem die gute Performance der adaptiven Taskpools unterstrichen. Nur durch den zusätzlichen Aufwand bei der initialen Balancierung können die konventionellen Taskpool-Implementierungen ihren geringeren Overhead beim Zugriff auf die eigenen Task-Datenstrukturen ausnutzen, aber die adaptiven Taskpools erreichen ohne diesen Aufwand deutliche bessere Resultate. Durch Einsatz der adaptiven Taskpools kann also auf eine bestimmte Anfangsverteilung verzichtet werden, da innerhalb weniger Operationen ein Großteil der Tasks auf die anderen Threads verteilt wird. Selbst der schlecht möglichste Fall der Speicherung aller initialen Tasks bei nur einem Thread kann von der adaptiven Datenstruktur gut gehandhabt werden.

7.7 Zusammenfassung

Die in diesem Kapitel vorgestellten Profiling-Mechanismen erlauben eine detaillierte Untersuchung einer taskbasierten Applikation. Probleme bei der Skalierbarkeit können aufgedeckt werden, ohne Codeänderungen vornehmen zu müssen. Die Ausführungszeit wird dabei jeweils getrennt für den Applikationskontext und den Taskpoolkontext protokolliert, sodass eine genaue Analyse einzelner Tasks möglich ist. Durch Betrachtung des Applikationskontextes können mögliche Skalierbarkeitsprobleme innerhalb der Taskstruktur entdeckt werden, wenn z. B. Locks die Laufzeit von Tasks ansteigen lassen oder generell eine ungewollte Abhängigkeit von der Anzahl benutzter Threads festgestellt wird.

Der Taskpoolkontext beschreibt andererseits die nötigen Berechnungen zur Verwaltung der Tasks und der Lastbalancierung. Diese gemessenen Zeiten sind Overhead und sollten entsprechend möglichst gering sein. Einerseits lässt sich dies durch effizientere Taskpool-Implementierungen realisieren, andererseits werden aber auch die Wartezeiten auf neue Tasks gemessen, sodass eine geringe Parallelität durch zu wenige ausführungsbereite Tasks festgestellt werden kann.

Ähnliche Profiling-Informationen sind auf anderem Wege nur durch Änderungen der Ap-

plikation oder durch entsprechende Instrumentierung des Compilers zu erhalten, die oftmals die parallele Ausführung begrenzen.

Die Informationen über die einzelnen Tasks erlauben auch eine Performance-Vorhersage für eine bestimmte Anzahl zu benutzender Prozessoren. Man kann abschätzen, ob bei der Benutzung zusätzlicher Prozessoren noch eine signifikante Verbesserung möglich ist. Außerdem gibt die prognostizierte Ausführungszeit Hinweise, welche Tasks aufgrund ihrer Anteile an der Gesamtlaufzeit am ehesten verbessert werden sollten.

Kapitel 8

Zusammenfassung und Ausblick

Die Ausführung irregulärer Applikationen mit feingranularen Tasks ermöglicht die effiziente Nutzung aktueller Parallelrechner mit einer großen Zahl eng gekoppelter Prozessoren bzw. Prozessorkerne. Zu Beginn dieser Arbeit wurde untersucht, wie sich der Verwaltungsaufwand für diese Tasks mit kürzerer Laufzeit verringern lässt, indem Hardware-Operationen zur Synchronisation eingesetzt werden. Diese Operationen erlauben einen effizienten kontrollierten Zugriff mehrerer Threads auf gemeinsame Datenstrukturen. Die Instruktionen können teilweise auch direkt genutzt werden, um Datenstrukturen ohne explizite Sperrmechanismen zu modifizieren. Da praktisch alle verfügbaren Prozessoren solche Synchronisationskonzepte in teilweise unterschiedlicher Ausprägung unterstützen, ist der Einsatz sinnvoll und notwendig, um die bestmögliche Performance zu erreichen. Ohne Veränderungen innerhalb der getesteten Applikationen vornehmen zu müssen, konnten teilweise erhebliche Verbesserungen festgestellt werden.

Innerhalb des vorgestellten KOALA-Frameworks stehen Implementierungen dieser Synchronisationsmechanismen zur Verfügung, es können aber für zukünftige Systeme entsprechend angepasste Komponenten hinzugefügt werden. Darüber hinaus bietet das KOALA-Framework eine in Schichten zerlegte Abstraktion der taskbasierten Programmabarbeitung. Durch austauschbare Komponenten mit festem Interface können unterschiedliche Algorithmen zur Lastbalancierung verwendet werden. Zur Implementierung kann zudem auf Hilfskomponenten zurückgegriffen werden, die für das zugrunde liegende System angepasste Funktionen bieten. Das vorgestellte KOALA-Framework besteht aus vier wesentlichen Komponenten, die Taskausführung, Taskspeicherung und Balancierung sowie Speicherverwaltung und Synchronisation getrennt implementieren. Ohne Einflussnahme des Programmierers werden in der Konfigurationsphase die für das betrachtete System verfügbaren Komponenten aktiviert, sodass funktionierende Implementierungen aller Bestandteile benutzt werden können, ohne genaue Kenntnis des Systems oder des Frameworks haben zu müssen. Bei Bedarf können jedoch spezielle Komponenten ausgewählt werden, um spezifische Untersuchungen durchführen zu können. Eine automatische Tuning-Komponente hilft dem Programmierer, die für eine effiziente Ausführung wesentlichen Parameter systemabhängig festzulegen.

Die Wahl der zu verwendenden Taskpool-Implementierung kann entscheidend für das Erreichen einer guten Performance sein. Die betrachteten konventionellen Taskpool-Implementierungen wurden detailliert untersucht, können aber teilweise nur unter gewissen Bedingungen gute Resultate erreichen. Daher wurde in dieser Arbeit ein neuartiger adaptiver Taskpool entwickelt. Die verwendete Datenstruktur erlaubt es, eine große Anzahl von Tasks derart zu speichern, dass einerseits der Zugriff beim Einfügen oder Entnehmen nicht

wesentlich langsamer ist als bei z. B. verketteten Listen, aber andererseits eine Lastbalancierung in wenigen Schritten möglich ist. Das zur Lastbalancierung angewendete Task-Stealing kann in Abhängigkeit von der Anzahl der gespeicherten Tasks unterschiedlich große Blöcke von Tasks mit einer einzelnen Operation entnehmen. Es wurde gezeigt, dass es eine untere Schranke für die Anzahl der in einem Schritt entnommenen Tasks gibt, die z. B. bei binären Bäumen bei einem Viertel aller gespeicherten Tasks liegt. Laufzeittests mit irregulären Applikationen mit unterschiedlicher Taskstruktur zeigten die gute Performance dieser Taskpool-Implementierung. Die adaptiven Taskpools konnten in allen betrachteten Fällen gute Laufzeiten erzielen, wogegen teilweise andere Taskpool-Implementierungen bei manchen Lastzuständen erheblich schlechtere Resultate erzielten. Die adaptiven Taskpools eignen sich daher besonders als allgemeiner Ansatz für die effiziente Verwaltung einer großen Anzahl von Tasks. Zwar können für eine Applikation spezialisierte Taskpool-Implementierungen bessere Laufzeiten erzielen, die adaptiven Taskpools können aber auch ohne genaues Wissen über die Taskstruktur der Applikation und deren mögliches Laufzeitverhalten eingesetzt werden. Dabei erzielen sie mindestens ähnlich gute Resultate wie die konventionellen Taskpool-Implementierungen und es gab bei den Untersuchungen keine Applikation, bei denen die adaptiven Taskpools weniger gute Ergebnisse erzielten.

Unabhängig von der Taskpool-Implementierung ist die Taskstruktur der jeweiligen Applikation entscheidend für eine gute Skalierbarkeit. Daher wurde als alternatives Taskpool-Frontend eine Profiling-Komponente entwickelt, mit der eine genaue Analyse der einzelnen Tasktypen einer Applikation möglich ist. Im Gegensatz zu anderen Profiling-Mechanismen ist der verwendete Ansatz ohne Änderungen in der Applikation oder gar Neukompilierung mit Instrumentierung verwendbar, sodass auch Untersuchungen der parallelen Abarbeitung ohne signifikante Einflüsse möglich sind. Die gemessenen Daten können zur Bewertung der einzelnen Tasks verwendet werden, um besonders kritische Tasks zu identifizieren. Außerdem lässt sich mit einer Laufzeitvorhersage ein vermuteter Speedup abschätzen, der bei Einsatz einer bestimmten Anzahl von Prozessoren erreicht werden kann. Entsprechend der vorhergesagten Laufzeit können einzelne Tasks als besonders relevant für Verbesserungen identifiziert werden. Die Anwendung der gesammelten Informationen auf das hierarchische Radiosity-Verfahren konnte dessen Skalierbarkeit erheblich verbessern. Darüber hinaus konnten die adaptiven Taskpools die Verbesserungen der Applikation wesentlich besser umsetzen, da eine Lastbalancierung der initialen Tasks ohne großen Aufwand möglich ist. Die konventionellen Taskpool-Implementierungen konnten einen guten Speedup nur erreichen, indem zusätzlicher Aufwand zur Balancierung der initialen Tasks betrieben wurde. Auf eine gute Anfangsverteilung kann also beim Einsatz adaptiver Taskpools verzichtet werden.

Für zukünftige Untersuchungen haben sich während dieser Arbeit folgende Aspekte ergeben, die als weitere Forschungsansätze dienen können.

Während die adaptiven Taskpools in einer Hochsprache implementiert wurden, kann eine Hardware-nahe Implementierung erhebliche Verbesserungen bei der Verwaltung besonders kleiner Tasks bewirken. Insbesondere können effiziente Hardware-Operationen bei der Modifikation der adaptiven Datenstruktur den Overhead reduzieren. Zudem kann untersucht werden, ob die Unterteilung in öffentliche und private Bereiche bei der adaptiven Datenstruktur bei Hardware-naher Implementierung eventuell bessere Ergebnisse liefert, da der Aufwand für die Verwaltung verringert werden kann.

Die Messung der Ausführungszeiten und Wartezeiten einzelner Tasks wurde zur Analyse der Taskstruktur verwendet. Es ist aber auch denkbar, solche Informationen direkt während der Laufzeit auszuwerten und in Lastbalancierungsentscheidungen einfließen zu lassen. Insbesondere kann die adaptive Datenstruktur so modifiziert werden, dass nicht ausschließlich die Anzahl der gespeicherten Tasks die Größe der Teilbäume festlegt, sondern vielmehr der mit den Tasks verbundene prognostizierte Rechenaufwand. Es ist zu untersuchen, ob es dadurch möglich ist, dass ein gestohlener Teilbaum nicht ein Viertel aller Tasks sondern ein Viertel des Rechenaufwands der gespeicherten Tasks enthält und so eine effektivere Lastbalancierung anhand der Rechenlast realisiert werden kann.

Anhang A

Rechnersysteme

A.1 IBM eServer pSeries 690

Das System „JUMP“ des Jülicher Forschungszentrums existiert in zwei Ausbaustufen. Der Großteil der Messungen in dieser Arbeit für das IBM-System wurde auf der ersten Variante durchgeführt. Diese besitzt 32 Power4+-Prozessorkerne, Details dazu sind in Tabelle A.1 zusammengefasst. Im Verlauf der Arbeit wurden die Prozessoren durch 32 Power6-Prozessorkerne mit deutlich höherer Taktfrequenz ersetzt (Tabelle A.2).

Prozessortyp	Power4+
Anzahl Kerne	32
Taktfrequenz	1,7 GHz
Hauptspeicher	128 GB
Cache	L1:32+64 KB (Daten+Instruktionen)
	L2:1,5 MB für je 2 Kerne
	L3:32 MB für je 2 Kerne

Tabelle A.1: Technische Parameter des IBM p690 Systems (erste Ausbaustufe).

Prozessortyp	Power6
Anzahl Kerne	32 * 2 (SMT)
Taktfrequenz	4,7 GHz
Hauptspeicher	128 GB
Cache	L1:64+64 KB (Daten+Instruktionen)
	L2:4 MB
	L3:32 MB für je 2 Kerne

Tabelle A.2: Technische Parameter des IBM p690 Systems (zweite Ausbaustufe).

A.2 SGI Altix 4700

Der Supercomputer „SGI Altix 4700“ des Leibniz-Rechenzentrums München besitzt 4864 Dual-Core-Itanium2-Prozessoren, wobei bis zu 512 davon in einer Partition mit gemeinsamem Speicher genutzt werden können. Technische Details sind in Tabelle A.3 dargestellt.

Prozessortyp	Itanium2
Anzahl Kerne	512 (als SMP verfügbar)
Taktfrequenz	1,6 GHz
Hauptspeicher	4 GB pro Kern
Cache	L1:16 KB (Daten)
	L2:256+1024 KB (Daten+Instruktionen)
	L3:9 MB pro Kern

Tabelle A.3: Technische Parameter des SGI Altix 4700 Systems.

A.3 Sun Fire 6800

Das SPARC-System „Sun Fire 6800“ der Martin-Luther-Universität Halle-Wittenberg besteht aus 24 UltraSPARC-III-Prozessoren, deren technische Details in Tabelle A.4 abgebildet sind.

Prozessortyp	UltraSPARCIII+
Anzahl Kerne	24
Taktfrequenz	0,9 GHz
Hauptspeicher	24 GB
Cache	L1:64+32 KB (Daten+Instruktionen)
	L2:8 MB

Tabelle A.4: Technische Parameter des Sun Fire 6800 Systems.

Anhang B

Zugehörige Publikationen

- R. Hoffmann, M. Korch, and T. Rauber. Performance Evaluation of Task Pools Based on Hardware Synchronization. In *Proceedings of the 2004 Supercomputing Conference (SC'04)*, Pittsburgh, PA, USA, Nov. 2004. IEEE/ACM SIGARCH.
- R. Hoffmann, M. Korch, and T. Rauber. Using Hardware Operations to Reduce the Synchronization Overhead of Task Pools. In *Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)*, pages 241–249. IEEE Computer Society Press, 2004.
- R. Hoffmann and T. Rauber. Profiling of Task-based Applications on Shared Memory Machines: Scalability and Bottlenecks. In *Proceedings of Euro-Par 2007*, volume 4641 of *LNCS*, pages 118–128. Springer, 2007.
- R. Hoffmann, S. Hunold, M. Korch, and T. Rauber. Towards Scalable Parallel Numerical Algorithms and Dynamic Load Balancing Strategies. In *Proceedings of the Third Joint HLRB and KONWIHR Result and Reviewing Workshop 2007*, pages 503–516. Springer, 2008.
- R. Hoffmann and T. Rauber. Fine-Grained Task Scheduling Using Adaptive Data Structures. In *Proceedings of Euro-Par 2008*, volume 5168 of *LNCS*, pages 253–262. Springer, 2008.

Literaturverzeichnis

[1] A. Adl-Tabatabai, C. Kozyrakis, and B. Saha. Unlocking Concurrency. *ACM Queue*, 4(10):24–33, Dec 2006.

[2] O. Agesen, D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. L. Steele, Jr. DCAS-Based Concurrent Deques. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 137–146. ACM, 2000.

[3] K. Agrawal, Y. He, and C. E. Leiserson. Adaptive Work Stealing with Parallelism Feedback. In K. A. Yelick and J. M. Mellor-Crummey, editors, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (22th PPOPP'2007)*, pages 112–120. ACM, 2007.

[4] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele, Jr., and S. Tobin-Hochstadt. The Fortress Language Specification, version 1.0beta. Technical report, SUN, Mar. 2007.

[5] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence Based Approach*. Morgan Kaufman, 2002.

[6] J. H. Anderson, Y.-J. Kim, and T. Herman. Shared-memory Mutual Exclusion: Major Research Trends Since 1986. *Distributed Computing*, 16:75–100, 2003. Special issue celebrating the 20th anniversary of PODC.

[7] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Dep., Univ. of California, Berkeley, Dec. 2006.

[8] I. Banicescu, V. Velusamy, and J. Devaprasad. On the Scalability of Dynamic Scheduling Scientific Applications with Adaptive Weighted Factoring. *Cluster Computing, The Journal of Networks, Software Tools and Applications*, 6:215–226, 2003.

[9] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: A Programming Model for the Cell BE Architecture. In *Proceedings of the 2006 ACM/IEEE SC'06 Conference*. IEEE, 2006.

[10] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. *ACM SIGPLAN Notices*, 35(11):117–128, 2000.

[11] W. Blochinger and W. Küchlin. The Design of an API for Strict Multithreading in C++. In *Euro-Par 2003. Parallel Processing*, number 2790 in LNCS, pages 722–731. Springer, Aug. 2003.

[12] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming (PPOPP'1995*, pages 55–69. ACM, 1995.

[13] H.-J. Boehm. Fast Multiprocessor Memory Allocation and Garbage Collection. Technical Report HPL-2000-165, HP Labs, Dec. 2000.

[14] S. H. Bokhari. On the Mapping Problem. *IEEE Transactions on Computers*, 30(3):550–557, 1981.

[15] S. H. Bokhari. *Assignment Problems in Parallel and Distributed Computing.* Kluwer Academic Publishers, 1987.

[16] S. Y. Borkar, P. Dubey, K. C. Kahn, D. J. Kuck, H. Mulder, S. S. Pawlowski, and J. R. Rattner. Platform 2015: Intel Processor and Platform Evolution for the Next Decade. *Technology@Intel Magazine*, Mar. 2005.

[17] P. Brucker. *Scheduling Algorithms.* Springer, Berlin, 2004.

[18] H. Brunst, D. Kranzlmüller, and W. E. Nagel. Tools for Scalable Parallel Program Analysis - Vampir VNG and DeWiz. In *Distributed and Parallel Systems: Cluster and Grid Computing (DAPSYS 2004, Workshop on Distributed and Parallel Systems)*, pages 93–102. Springer, 2004.

[19] K. Burrage. *Parallel and Sequential Methods for Ordinary Differential Equations.* Oxford Science Publications, 1995.

[20] F. W. Burton and M. R. Sleep. Executing Functional Programs on a Virtual Tree of Processors. In *Proceedings of the 1981 Conference on Functional programming languages and computer architecture (FPCA '81)*, pages 187–194, New York, NY, USA, 1981. ACM.

[21] C. Busch, M. Mavronicolas, and P. Spirakis. The Cost of Concurrent, Low-Contention Read-Modify-Write. In *Proceedings of the 10th Colloquium on Structural Information and Communication Complexity (SIROCCO 2003)*, pages 57–72, Umee, Sweden, 2003. Carleton Scientific.

[22] D. R. Butenhof. *Programming with POSIX Threads.* Addison Wesley, 1997.

[23] J. A. Butts and G. S. Sohi. A Static Power Model for Architects. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 191–201, New York, NY, USA, 2000. ACM.

[24] R. L. Cariño and I. Banicescu. A Load Balancing Tool for Distributed Parallel Loops. In *Proceedings of the International Workshop on Challenges of Large Application in Distributed Environments (CLADE) 2003*, pages 39–46. IEEE, June 2003.

[25] W. W. Carlson, J. M. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, Center for Computing Sciences, IDA, Bowie, MD, May 1999.

[26] C. Chantrapornchai, S. Tongsima, and E. H.-M. Sha. Imprecise Task Schedule Optimization. In *Proceedings of the International Conference on Fuzzy Systems*, pages 1265–1270. IEEE, 1997.

[27] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In R. Johnson and R. P. Gabriel, editors, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 519–538. ACM, Oct. 2005.

[28] A. T. Chronopoulos, S. Penmatsa, and N. Yu. Scalable Loop Self-Scheduling Schemes for Heterogeneous Clusters. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'02)*, pages 353–359. IEEE, Nov. 2002.

[29] D. Callahan and B. L. Chamberlain and H. P. Zima. The Cascade High Productivity Language. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*, pages 52–60. IEEE, 2004.

[30] S. Dandamudi. *Hierarchical Scheduling in Parallel and Cluster Systems.* Kluwer Academic/Plenum Publishers, New York, 2003.

[31] T. Decker. Virtual Data Space – Load Balancing for Irregular Applications. *Parallel Computing*, 26(13–14):1825–1860, Dec. 2000.

[32] E. W. Dijkstra. Cooperating Sequential Processes. Technical report, Technological University, Eindhoven, 1965.

[33] J. Dongarra, J. London, S. Browne, S. Browne, J. Dongarra, N. Garner, N. Garner, K. London, P. Mucci, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14:189–204, 2000.

[34] T. H. Dunigan, J. S. Vetter, and P. H. Worley. Performance Evaluation of the SGI Altix 3700. In *Proceedings of the 2005 International Conference on Parallel Processing (34th ICPP'2005)*, pages 231–240. IEEE, June 2005.

[35] A. E. Eichenberger, K. M. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing Compiler for the CELL Processor. In *Proceedings of the 14th International Conference Parallel Architectures and Compilation Techniques*, pages 161–172. IEEE, 2005.

[36] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared Memory Programming.* Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, Hoboken, New Jersey, June 2005.

[37] R. Elsässer and B. Monien. Load Balancing of Unit Size Tokens and Expansion Properties of Graphs. In *Proceedings of the 15th Annual ACM symposium on Parallel algorithms and architectures (SPAA '03)*, pages 266–273, New York, NY, USA, 2003. ACM.

[38] N. Faroughi. Multi-Cache Profiling of Parallel Processing Programs Using Simics. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications & Conference on Real-Time Computing Systems and Applications, PDPTA 2006, Las Vegas, Nevada, USA, June 26-29, 2006, Volume 1*, pages 499–505. CSREA Press, 2006.

[39] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE SC'06 Conference.* IEEE, 2006.

[40] T. L. Freeman, D. J. Hancock, J. M. Bull, and R. W. Ford. Feedback Guided Scheduling of Nested Loops. In T. Sorevik, F. Manne, R. Moe, and A. H. Gebremedhin, editors, *Applied Parallel Computing*, number 1947 in LNCS, pages 149–159. Springer, 2001.

[41] S. Frenz, M. Schoettner, R. Goeckelmann, and P. Schulthess. Transactional Cluster Computing. In *Proceedings of the 2005 International Conference on High Performance Computing and Communications (HPCC)*, pages 465–476. Springer, 2005.

[42] A. Gerasoulis, J. Jiao, and T. Yang. Experience with Graph Scheduling for Mapping Irregular Scientific Computation. In *Proceedings of the 1st IPPS Workshop on Solving Irregular Problems on Distributed Memory Machines.* IEEE, Apr. 1995.

[43] A. Gerasoulis and T. Yang. Static Scheduling of Parallel Programs for Message Passing Architectures. In L. Bouge, M. Cosnard, Y. Robert, and D. Trystram, editors, *Proceedings of the Second Joint International Conference on Vector and Parallel Processing. Parallel Processing: VAPP V, CONPAR'92*, number 634 in LNCS, pages 601–612. Springer, 1992.

[44] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice.* Addison-Wesley, 2006.

[45] M. Griebel, S. Knapek, G. Zumbusch, and A. Caglar. *Numerische Simulation in der Moleküldynamik. Numerik, Algorithmen, Parallelisierung, Anwendungen.* Springer, Berlin, Heidelberg, 2004.

[46] M. Griebl and C. Lengauer. The Loop Parallelizer LooPo. In M. Gerndt, editor, *Proceedings of the 6th Workshop on Compilers for Parallel Computers*, volume 21 of *Konferenzen des Forschungszentrums Jülich*, pages 311–320. Forschungszentrum Jülich, 1996.

[47] R. H. Halstead, Jr. Implementation of Multilisp: Lisp on a Multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming (LFP '84)*, pages 9–17, New York, NY, USA, 1984. ACM.

[48] P. Hanrahan, D. Salzman, and L. Aupperle. A Rapid Hierarchical Radiosity Algorithm. *ACM SIGGRAPH Computer Graphics*, 25(4):197–206, July 1991.

[49] R. W. Hay and G. R. Hook. POWER4 and Shared Memory Synchronisation. Technical report, IBM, Apr. 2002.

[50] J. Held and J. B. ans S. Koehl. From a Few Cores to Many – A Tera-Scale Computing Research Overview. Intel White Paper, Intel, 2006.

[51] D. Hendler and N. Shavit. Non-Blocking Steal-Half Work Queues. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing (PODC'02)*, pages 280–289. ACM, 2002.

[52] J. L. Hennessy and D. A. Patterson. *Computer Architecture — A Quantitative Approach.* Morgan Kaufmann, 2007.

[53] M. Herlihy, V. Luchangco, and M. Moir. Space- and Time-Adaptive Nonblocking Data Structures. In *Computing: The Australasian Theory Symposium (CATS)*, pages 260–280. Elsevier, 2003.

[54] M. Herlihy and J. Moss. Transactional Memory: Architectural Support for Lock-free Data Stractures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*, pages 289–300. IEEE, 1993.

[55] High Performance Fortran Forum. *High Performance Fortran Language Specification, version 2.0.* Center for Research on Parallel Computation, Rice Univ., Houston, TX, Jan. 1997.

[56] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(4):10–15, 1962.

[57] R. Hoffmann. Reduzierung des Synchronisations- und Verwaltungsoverheads von Taskpools auf SharedMemory-Systemen. Diplomarbeit, Martin-Luther-Universität Halle-Wittenberg, Apr. 2003.

[58] R. Hoffmann, M. Korch, and T. Rauber. Performance Evaluation of Task Pools Based on Hardware Synchronization. In *Proceedings of the 2004 Supercomputing Conference (SC'04)*, Pittsburgh, PA, Nov. 2004. IEEE/ACM SIGARCH.

[59] H. P. Hofstee and M. Day. Hardware and Software Architectures for the CELL Processor. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '05)*, page 1. ACM, 2005.

[60] Intel Corporation. Intel Itanium Processor Reference Manual for Software Optimization. Technical report, Intel Corporation, Nov. 2001.

[61] Intel Corporation. *Intel IA-64 Architecture Software Developer's Manual: Volume 2: IA-64 System Architecture*. Intel Corporation, Oct. 2002.

[62] Intel Corporation. *IA-32 Intel Architecture Optimization Reference Manual*. Intel Corporation, Apr. 2006.

[63] K. Jansen and H. Zhang. An Approximation Algorithm for Scheduling Malleable Tasks under General Precedence Constraints. *ACM Transactions on Algorithms (TALG)*, 2(3):416–434, 2006.

[64] E. H. Jensen, G. W. Hagensen, and J. M. Broughton. A New Approach to Exclusive Data Access in Shared Memory Multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, Livermore, CA, Nov. 1987.

[65] L. V. Kalé and S. Krishnan. CHARM++. In G. V. Wilson and P. Lu, editors, *Parallel Programming in C++*, chapter 5, pages 175–214. MIT Press, Cambridge, MA, 1996.

[66] G. Karypis and V. Kumar. METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Version 4.0. Technical report, University of Minnesota, Sept. 1998.

[67] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive Performance and Scalability Modeling of a Large-Scale Application. In *Proceedings of the 2001 Supercomputing Conference (SC'01)*, page 37. IEEE/ACM SIGARCH, 2001.

[68] B. W. Kernighan and D. M. Richie. The M4 Macro Processor. In *Unix Programmer's Manual*. Bell Laboratories, Murray Hill, NJ, 1979.

[69] M. Korch and T. Rauber. A Comparison of Task Pools for Dynamic Load Balancing of Irregular Algorithms. *Concurrency and Computation: Practice and Experience*, 16:1–47, Jan. 2004.

[70] M. Kowarschik, U. Rüde, N. Thürey, and C. Weiß. Performance Optimization of 3D Multigrid on Hierarchical Memory Architectures. In *Proceedings of the 2002 Conference on Applied Parallel Computing (PARA'02)*, number 2367 in LNCS, pages 307–318, Espoo, Finland, June 2002. Springer.

[71] S. Krishnamoorthy, U. Catalyurek, J. Nieplocha, A. Rountev, and P. Sadayappan. Hypergraph Partitioning for Automatic Memory Hierarchy Management. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC'06)*, page 98. ACM, 2006.

[72] W. Kuchera and C. Wallace. The UPC Memory Model: Problems and Prospects. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, page 16a. IEEE, 2004.

[73] M. Kühnemann, T. Rauber, and G. Rünger. Source Code Analyzer for Cost Modeling of Parallel Application Programs. In *Proceedings of the 4th IPDPS-Workshop on Massively Parallel Processing*, page 262. IEEE, 2004.

[74] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. *ACM SIGARCH Computer Architecture News*, 35(2):162–173, May 2007.

[75] S. Kumar, D. Jiang, J. P. Singh, and R. Chandra. Evaluating Synchronization on Shared Address Space Multiprocessors: Methodology and Performance. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computing Systems (SIGMETRICS-99)*, pages 23–34, New York, May 1999. ACM.

[76] V. Kumar, A. Y. Grama, and N. R. Vempaty. Scalable Load Balancing Techniques for Parallel Computers. *Journal of Parallel and Distributed Computing*, 22(1):60–79, 1994.

[77] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, pages 241–251, New York, June 1997. ACM.

[78] D. Lea. A Memory Allocator. *Unix/Mail*, 6/96, 1996.

[79] R. Lepère, D. Trystram, and G. Woeginger. Approximation Scheduling For Malleable Tasks under Precedence Constraints. *International Journal of Foundation in Computer Science*, 13(4):613, 2002.

[80] J. Y.-T. Leung, editor. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman & Hall/CRC, Boca Raton, FL, 2004.

[81] M. Levoy. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics*, 9(3):245–251, July 1990.

[82] M. Levoy. Volume Rendering by Adaptive Refinement. *The Visual Computer*, 6(1):2–7, Feb. 1990.

[83] V. Luchangco, M. Moir, and N. Shavit. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *Proceedings of the 23th International Conference on Distributed Computing Systems (23th ICDCS'03)*, pages 522–529. IEEE, 2003.

[84] R. Lüling and B. Monien. Load Balancing for Distributed Branch and Bound Algorithm. In *Proceedings of 6th International Parallel Processing Symposium*, pages 543–548. IEEE, March 1992.

[85] R. Lüling and B. Monien. A Dynamic Distributed Load Balancing Algorithm with Provable Good Performance. In *Proceedings of 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 164–172. ACM, 1993.

[86] A. Malony, S. S. Shende, and A. Morris. Phase-Based Parallel Performance Profiling. In *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, volume 33 of *John von Neumann Institute for Computing Series*, pages 203–210. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[87] G. Marin and J. Mellor-Crummey. Cross-Architecture Performance Predictions for Scientific Applications Using Parameterized Models. In *Proceedings of Joint International Conference on Measurement and Modeling of Computer Systems - Sigmetrics 2004*, pages 2–13, New York, NY, USA, June 2004. ACM.

[88] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. `http://www.cs.rochester.edu/u/scott/synchronization/pseudocode/ss.html`.

[89] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[90] H. Meyerhenke and S. Schamberger. A Parallel Shape Optimizing Load Balancer. In W. E. Nagel, W. V. Walter, and W. Lehner, editors, *Euro-Par 2006. Parallel Processing*, number 4128 in LNCS, pages 232–242. Springer, 2006.

[91] M. M. Michael and M. L. Scott. Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[92] B. Monien and S. Schamberger. Graph Partitioning with the Party Library: Helpful-Sets in Practice. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04)*, pages 198–205. IEEE, 2004.

[93] MPI Forum. MPI: A Message-Passing Interface Standard, Version 1.1. Technical report, University of Tennessee, June 1995.

[94] MPI Forum. MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, July 1997.

[95] P. J. Mucci. Papiex. `http://icl.cs.utk.edu/~mucci/papiex/`.

[96] R. Nagel and T. Rauber. RCM – A Multi-layered Reconfigurable Cluster Middleware. In *Proceedings of the 14th Euromicro Conference on Parallel, Distributed and Network based Processing (PDP'06)*, pages 203–210, Montbeliard-Sochaux, France, February 2006. IEEE.

[97] G. J. Narlikar and G. E. Blelloch. Pthreads for Dynamic and Irregular Parallelism. In *Proceedings of the 1998 Supercomputing Conference (SC'98)*, page 1. IEEE, Nov. 1998.

[98] J. Nieh and M. Levoy. Volume Rendering on Scalable Shared-Memory MIMD Architectures. In *Proceedings of the Boston Workshop on Volume Visualization*, pages 17–24. ACM, Oct. 1992.

[99] S. Oaks and H. Wong. *Java Threads*. O'Reilly, 2nd edition, Jan. 1999.

[100] L. Oliker and R. Biswas. PLUM: Parallel Load Balancing for Adaptive Unstructured Meshes. *Journal of Parallel and Distributed Computing*, 52(2):150–177, Aug. 1998.

[101] K. Olukotun and L. Hammond. The Future of Microprocessors. *ACM Queue*, 3(7):26–29, 2005.

[102] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 2.5*, May 2005.

[103] C. Polychronopoulos, N. Bitar, and S. Kleiman. Nanothreads: A User-Level Threads Architecture. Technical Report 1297, CSRD, Univ. of Illinois at Urbana-Champaign,, 1993.

[104] C. Polychronopoulos and D. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, Dec. 1987.

[105] Power Architecture editors, developerWorks, IBM. Just like being there: Papers from the Fall Processor Forum 2005: Unleashing the power of the Cell Broadband Engine – A programming model approach. *IBM developerWorks*, Nov. 2005.

[106] J. Protič, M. Tomaševič, and V. Milutinovič, editors. *Distributed Shared Memory: Concepts and Systems*. IEEE, 1998.

[107] K. H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, MIT Dep. of Electrical Engineering and Computer Science, June 1998.

[108] T. Rauber and G. Rünger. *Parallele und verteilte Programmierung*. Springer Verlag, 2000.

[109] T. Rauber and G. Rünger. *Parallele Programmierung, 2. Auflage*. Springer Verlag, Heidelberg, 2007.

[110] L. Rauchwerger. Run-Time Parallelization: It's Time Has Come. *Journal of on Parallel Computing, Special Issue on Languages & Compilers for Parallel Computers*, 24(3–4):527–556, 1998.

[111] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 2007.

[112] M. S. Schlansker and B. R. R. Cover. EPIC: Explicitly Parallel Instruction Computing. *Computer*, 33(2):37–45, Feb. 2000.

[113] K. Schloegel, G. Karypis, and V. Kumar. A Unified Algorithm for Load-Balancing Adaptive Scientific Simulations. In *Proceedings of Supercomputing'2000*, pages 75–75. IEEE, 2000.

[114] M. Schulz, J. Tao, C. Trinitis, and W. Karl. SMiLE: An Integrated, Multi-Paradigm Software Infrastructure for SCI-based Clusters. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, page 247. IEEE, 2002.

[115] J. P. Schulze and U. Lang. The Parallelized Perspective Shear-Warp Algorithm for Volume Rendering. *Parallel Computing*, 29:339–354, 2003.

[116] J. Seward et al. Valgrind. `http://valgrind.org/`.

[117] M. Shah, J. Barren, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, B. Sana, D. Sheahan, L. Spracklen, and A. Wynn. UltraSPARC T2: A highly-treaded, power-efficient, SPARC SOC. In *IEEE Asian Solid-State Circuits Conference (A-SSCC) 2007*, pages 22–25. IEEE, 2007.

[118] J. P. Singh, A. Gupta, and M. Levoy. Parallel Visualization Algorithms: Performance and Architectural Implications. *IEEE Computer*, 27(7):45–55, July 1994.

[119] J. P. Singh, C. Holt, T. Tosuka, A. Gupta, and J. L. Hennessy. Load Balancing and Data Locality in Adaptive Hierarchical N-body Methods: Barnes-Hut, Fast Multipole, and Radiosity. *Journal of Parallel and Distributed Computing*, 27(2):118–141, June 1995.

[120] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, Mar. 1992.

[121] J. Singler, P. Sanders, and F. Putze. MCSTL: The Multi-Core Standard Template Library. In *Euro-Par 2007. Parallel Processing*, number 4641 in LNCS, pages 682–694. Springer, Aug. 2007.

[122] S. Spach and R. Pulleyblank. Parallel Raytraced Image Generation. *Hewlett-Packard Journal*, 43(3):76–83, June 1992.

[123] J. Stoer. *Numerische Mathematik 1*. Springer-Verlag, Berlin, 8th edition, 1999.

[124] J. M. Tendler, J. S. Dodson, J. S. Fields, Jr., H. Le, and B. Sinharoy. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, Jan. 2002.

[125] X. Tian, Y.-K. Chen, M. Girkar, S. Ge, R. Lienhart, and S. Shah. Exploring the Use of Hyper-Threading Technology for Multimedia Applications with Intel OpenMP Compiler. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS'2003)*, page 36. IEEE, 2003.

[126] S. Tongsima, C. Chantrapornchai, E. H.-M. Sha, and N. Passos. Probabilistic Rotation: Scheduling Graphs with Uncertain Execution Time. In *Proceedings of the International Conference on Parallel Processing*, pages 292–297. IEEE, 1997.

[127] S. Tschöke, R. Lüling, and B. Monien. Solving the Traveling Salesman Problem with a Distributed Branch-and-Bound Algorithm on a 1024 Processor Network. In *Proceedings of 9th International Parallel Processing Symposium*, pages 182–189. IEEE, 1995.

[128] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403. ACM, June 1995.

[129] J. Turek, J. Wolf, and P. Yu. Approximate Algorithms for Scheduling Parallelizable Tasks. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 323–332. ACM, 1992.

[130] V.-Y. Vee and W.-J. Hsu. A Scalable and Efficient Storage Allocator on Shared-Memory Multiprocessors. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN'99)*, pages 230–235, Fremantle, Australia, 1999. IEEE.

[131] I. E. Venetis and T. S. Papatheodorou. Tying Memory Management to Parallel Programming Models. In W. E. Nagel, W. V. Walter, and W. Lehner, editors, *Euro-Par 2006. Parallel Processing*, number 4128 in LNCS, pages 666–675. Springer, 2006.

[132] K.-P. Vo. Vmalloc: A General and Efficient Memory Allocator. *Software Practice & Experience*, 26:1–18, 1996.

[133] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, 2001.

[134] D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, Englewood Cliffs, New Jersey, 2000.

[135] V. Weaver and I.-C. Li. Cachetool. `http://www.csl.cornell.edu/~vince/projects/cachetool/`.

[136] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, 1995. ACM.

[137] M. Wu and X.-F. Li. Task-pushing: a Scalable Parallel GC Marking Algorithm without Synchronization Operations. In *Proceedings of the 21th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, pages 1–10. IEEE, Mar. 2007.

[138] C. Xu and F. C. Lau. *Load Balancing in Parallel Computers: Theory and Practice*. Kluwer Academic Publishers, 1997.

[139] T. Yang and C. Fu. Space/Time-Efficient Scheduling and Execution of Parallel Irregular Computations. *ACM Transactions on Programming Languages and Systems*, 20(6):1195–1222, Nov. 1998.

[140] T. Yang and A. Gerasoulis. PYRROS: Static Task Scheduling and Code Generation for Message Passing Multiprocessors. In *Proceedings of the 6th ACM International Conference on Supercomputing*, pages 428–437, Washington D.C., July 1992. ACM.

[141] Z. Yu and W. Shi. An Adaptive Rescheduling Strategy for Grid Workflow Applications. In *Proceedings of the 21th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, pages 1–8. IEEE, Mar. 2007.