Effiziente parallele Ausführung irregulärer rekursiver Programme

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

der Fakultät für Informatik

der Universität Karlsruhe (Technische Hochschule)

genehmigte

Dissertation

von

Stefan U. Hänßgen

aus Esslingen am Neckar

Tag der mündlichen Prüfung:	21. April 1998
Erster Gutachter:	Prof. Dr. Walter F. Tichy
Zweiter Gutachter:	Prof. Dr. Werner Zorn

Zusammenfassung

Durch die rasante Entwicklung der Mehrprozessortechnologie ist der „Parallelrechner auf jedem Schreibtisch“ absehbar. Gegenüber den Hardware-Fortschritten stellt die Programmierung solcher Rechner aber immer noch ein Problem dar. Wünschenswert ist eine maschinenunabhängige, leicht zu nutzende, explizit parallele Sprache oder aber die automatische Parallelisierung von Programmen, auf die ich mich in dieser Arbeit konzentriere.

Trotz großer Fortschritte auf diesem Gebiet entziehen sich viele Bereiche nach wie vor der automatischen Parallelisierung oder zumindest der effizienten parallelen Ausführung. Eine Klasse solcher Programme zeichnet sich durch rekursive Berechnungen aus, deren nicht statisch vorhersehbares, datenabhängiges und unregelmäßiges Ablaufverhalten herkömmliche parallelisierende Übersetzer scheitern läßt. Eine Vielzahl naturwissenschaftlicher, mathematischer oder grafischer Probleme fallen in diesen Bereich, unter anderem auch zahlreiche Teile-und-Herrsche Algorithmen.

Methoden und Werkzeuge, die sich automatisch den potentiellen Parallelismus unabhängiger Rekursionsäste in solchen Programmen zunutze machen, sind daher erstrebenswert.

Meine Arbeit leistet auf diesem Gebiet die folgenden Beiträge:

Parallelisierungsstrategien: Es werden effiziente Parallelisierungsstrategien speziell für irreguläre rekursive Programme entworfen, die anstelle rekursiver Aufrufe parallel ablaufende Threads erzeugen. Zielarchitektur sind Parallelrechner mit gemeinsamem Speicher.

Automatische Parallelisierung: Das im Rahmen der Arbeit entwickelte REAPAR-System parallelisiert in ANSI C geschriebene Programme automatisch durch Quellcodetransformation zur Verwendung dieser Strategien.

Automatische Datensammlung: Relevante Laufzeitdaten des Programms werden durch eine automatische Instrumentierung des Quellcodes aufgezeichnet.

Automatische Strategiewahl: Das REAPAR-System analysiert automatisch die aufgezeichneten Daten und wählt daraufhin eine für das Problem geeignete Parallelisierungsstrategie durch eine Kombination von Threadablauf-Simulation und Heuristiken.

Die Wirksamkeit des Systems wurde anhand von neun in sich geschlossenen Benchmark-Anwendungen mit 190 bis 2 500 LOC validiert. Die Ergebnisse sind:

- Abhängig von Programm und Eingabedaten erreicht das REAPAR-System eine Beschleunigung zwischen 2,8 und 4 auf vier Prozessoren einer SPARCstation 20.
- Die automatische Strategiewahl selektiert fast immer die Strategie, die in der Realität am besten abschneidet. Sie erreicht mindestens 80% der optimalen Leistung, im Mittel der Benchmarks 93%.
- In Gegenüberstellung mit publizierten Beschleunigungen erreicht das REAPAR-System die Leistung vergleichbarer Systeme, die auf manuellen Parallelisierungen basieren, oder übertrifft sie.
- Eine Beschleunigung von bis zu 33 (sic!) auf 30 Prozessoren belegt die Skalierbarkeit der Verfahren für grobkörnige Probleme.
- Durch eine Kontrollmenge von vier Benchmarks wird die Leistung des Systems für bisher unbekannte Programme nachgewiesen, d.h. die entwickelten Methoden sind allgemeingültig.
- Eine Fallstudie zeigt, daß das System im Gegensatz zu einer Handparallelisierung für die automatische Parallelisierung und Strategiewahl nur einen verschwindenden Bruchteil der Arbeitszeit benötigt, ähnliche Ergebnisse erzielt und dabei keine weitergehenden Kenntnisse der Parallelrechnerprogrammierung voraussetzt.

Inhaltsverzeichnis

1 Einführung **1**
- 1.1 Motivation . . . 1
- 1.2 Stand der Forschung . . . 2
- 1.3 Ziel . . . 2
- 1.4 Überblick . . . 3
- 1.5 Beiträge dieser Arbeit . . . 3
- 1.6 Zusammenfassung der Ergebnisse . . . 5
  - 1.6.1 Beschleunigungen im Literaturvergleich . . . 5
  - 1.6.2 Qualität der automatischen Strategiewahl . . . 5
  - 1.6.3 Beschleunigungen auf mehr Prozessoren . . . 5

2 Grundlagen **6**
- 2.1 Parallelrechner . . . 6
  - 2.1.1 Architekturen . . . 6
    - 2.1.1.1 SIMD und MIMD . . . 6
    - 2.1.1.2 Gemeinsamer und verteilter Speicher . . . 7
  - 2.1.2 Maschinenmodelle . . . 8
  - 2.1.3 Herausforderungen . . . 8
    - 2.1.3.1 Datenabhängigkeiten . . . 8
    - 2.1.3.2 Datenlokalität und Caches . . . 9
    - 2.1.3.3 Lastverteilung . . . 9
    - 2.1.3.4 Granularität . . . 9
    - 2.1.3.5 Programmierung . . . 10
    - 2.1.3.6 Fazit . . . 10
- 2.2 Threads . . . 10
- 2.3 Rekursive Programme . . . 11
  - 2.3.1 Begriffsbestimmungen . . . 12
  - 2.3.2 Parallelitätspotential . . . 14
- 2.4 Unregelmäßigkeit . . . 14

3 Verwandte Arbeiten **16**
- 3.1 Explizit parallele Sprachen . . . 16
- 3.2 Automatische Parallelisierung . . . 17
- 3.3 Parallelisierung „dynamischer Programme“ . . . 18
- 3.4 Spezielle Ansätze für Teile-und-Herrsche . . . 20
- 3.5 Parallelisierung existierender Programme und Infrastruktur . . . 20
- 3.6 Einbeziehen von Laufzeitdaten . . . 21
- 3.7 Parallele funktionale und logische Sprachen . . . 21

3.8 Weitere Arbeiten . . . 22
3.9 Vergleich der erzielten Beschleunigungen . . . 22
3.10 Einordnung und Vergleich . . . 23

4 Anforderungen und Entwurf **25**
4.1 Anforderungen . . . 25
4.2 Annahmen . . . 26
4.3 Entwurf . . . 27
4.3.1 Parallelisierungsstrategien für rekursive Programme . . . 27
4.3.1.1 Parallelisierbarkeit . . . 28
4.3.1.2 Parallelisierungsstrategien . . . 29
4.3.2 Strategiewahl anhand von Laufzeitgrößen . . . 32
4.3.2.1 Kenngrößen . . . 32
4.3.2.2 Strategiewahl . . . 34
4.3.3 Instrumentierung zur Datengewinnung . . . 35
4.3.3.1 Anforderungen und Entwurfsentscheidungen . . . 35
4.3.3.2 Optionen . . . 37
4.3.4 Parallelisierung für Threads . . . 37
4.3.4.1 Anforderungen und Entwurfsentscheidungen . . . 37
4.3.4.2 Optionen . . . 37
4.4 REAPAR Systemüberblick . . . 38

5 Realisierung **40**
5.1 Szenario der Anwendung . . . 40
5.2 Techniken und Werkzeuge . . . 41
5.3 Annotationen . . . 41
5.4 Instrumentierungskomponente . . . 43
5.4.1 Instrumentierung . . . 43
5.4.2 Prinzip . . . 43
5.4.3 Algorithmus . . . 44
5.4.4 Baumaufzeichnung . . . 44
5.4.5 Beispielsausgabe . . . 46
5.4.6 Implementierung . . . 49
5.5 Parallelisierungskomponente . . . 49
5.5.1 Prinzip . . . 49
5.5.2 Algorithmus . . . 50
5.5.3 Implementierung . . . 52
5.6 Strategiewahlkomponente . . . 52
5.6.1 Datenanalyse und Strategiewahl . . . 52
5.6.2 Profilauswertung . . . 53
5.6.2.1 Prinzip . . . 53
5.6.2.2 Algorithmus . . . 54
5.6.2.3 Beispielausgabe . . . 56
5.6.2.4 Implementierung . . . 56
5.6.3 Simulation . . . 57
5.6.3.1 Prinzip . . . 57
5.6.3.2 Algorithmus . . . 58
5.6.3.3 Implementierung . . . 59

5.6.3.4 Beispiel . . . 61
5.6.3.5 Strategiewahl . . . 61
5.6.3.6 Wahl von sequentiellen Rekursionshierarchien . . . 64
5.7 Voraussetzungen und Einschränkungen . . . 64

6 Ergebnisse **67**
6.1 Benchmark-Suite . . . 67
6.1.1 Barnes Hut . . . 67
6.1.2 Eigenvalue . . . 69
6.1.3 Fractal . . . 71
6.1.4 Power . . . 72
6.1.5 Queens . . . 74
6.1.6 Beschreibung der Kenngrößen . . . 76
6.1.7 Tabelle der Kenngrößen . . . 76
6.2 Thesen, Experimente und Resultate . . . 78
6.2.1 Effiziente Beschleunigung irregulärer rekursiver Programme . . . 79
6.2.1.1 Versuchsaufbau und Messungen . . . 79
6.2.1.2 Erklärung der Meßgraphen . . . 79
6.2.1.3 Diskussion einer Meßreihe . . . 80
6.2.1.4 Erreichte Beschleunigungen . . . 82
6.2.1.5 Vergleich mit der Literatur . . . 84
6.2.2 Qualität der Tiefenheuristik . . . 85
6.2.3 Realitätsnähe der Simulation . . . 85
6.2.4 Nichtexistenz einer universellen Strategie . . . 87
6.2.5 Gültigkeit der Strategiewahl für andere Eingaben . . . 87
6.2.6 Effizienz durch geeignete Auswahl der Hierarchieebenen . . . 91
6.2.7 Effizienzgewinn durch Nothread-Annotation . . . 92
6.2.8 Kosten der Datensammlung . . . 93
6.2.9 Aufwand der Quellcodetransformationen . . . 94
6.2.10 Aufwand der Strategiewahl . . . 95
6.2.10.1 Tiefenwahl-Heuristik . . . 95
6.2.10.2 Simulation . . . 95
6.2.11 Kombination von Strategien . . . 96
6.3 Benchmarkläufe mit mehr Prozessoren . . . 97
6.3.1 Beschleunigungen . . . 98
6.3.2 Strategiewahl . . . 99
6.3.2.1 Heuristik . . . 99
6.3.2.2 Simulation . . . 100
6.4 Fallstudie . . . 101

7 Validierung **103**
7.1 Bemerkungen . . . 103
7.1.1 Skalierbarkeit . . . 103
7.1.2 Anpassung für REAPAR . . . 103
7.2 Bitonic Sort . . . 104
7.2.1 Beschreibung . . . 104
7.2.2 Beschleunigungen . . . 105
7.2.3 Strategiewahl . . . 107

7.3 Knapsack . . . 107
7.3.1 Beschreibung . . . 107
7.3.2 Beschleunigungen . . . 109
7.3.3 Strategiewahl . . . 109
7.4 Magic . . . 111
7.4.1 Beschreibung . . . 111
7.4.2 Beschleunigungen . . . 112
7.4.3 Strategiewahl . . . 114
7.5 Heat . . . 114
7.5.1 Beschreibung . . . 114
7.5.2 Beschleunigungen . . . 115
7.5.3 Strategiewahl . . . 117
7.6 Fazit . . . 117

8 Zusammenfassung und Ausblick **118**
8.1 Ergebnisse der vorliegenden Arbeit . . . 118
8.2 Weitere Richtungen der Forschung . . . 119

A **121**

Anhang **121**
A.1 Hilfsprogramme und Werkzeuge . . . 121
A.2 Versuche zum Maschinenlernen . . . 122
A.3 Versuche zur Threadzahl-Vorhersage . . . 124
A.4 Korrespondenz von Rekursionsbaum und Bildausschnitt bei Fractal . . . 125

Literaturverzeichnis **127**

Kapitel 1

Einführung

1.1 Motivation

Parallelrechner waren lange Zeit eine Domäne des „Supercomputing“, Werkzeuge für die Jagd nach immer mehr Rechenleistung für immer größere Probleme. Die Entwicklung der letzten Jahre gibt diesem Ansatz der Leistungssteigerung recht — in der Top 500 Liste der Supercomputer findet sich heute kein einziger Rechner mit weniger als drei Prozessoren [28]. Gleichzeitig hat sich durch immer kostengünstigere Herstellungstechniken der Anwendungsbereich von Parallelrechnern enorm ausgeweitet. Leistungsfähige Server-Rechner sind inzwischen grundsätzlich als Mehrprozessorsysteme ausgelegt, und der Schritt zum Parallelrechner auf jedem Schreibtisch ist absehbar.

Gegenüber diesen eindrucksvollen Fortschritten auf Seiten der Hardware stellt die Programmierung von Parallelrechnern immer noch ein Problem dar. Es gibt keine allgemeingültigen Verfahren, die eine gute Rechnerauslastung und eine optimale Beschleunigung von Programmen garantieren. War es zu Zeiten geringer Verbreitung noch akzeptabel, daß nur Spezialisten leistungsfähige Programme für Parallelrechner erstellen konnten, ist heute eine maschinennahe Programmierung nicht mehr angemessen — zu zahlreich sind die Fallstricke und zu wenig sind diese Programme portabel. Wünschenswert ist vielmehr eine maschinenunabhängige, leicht zu nutzende explizit parallele Sprache oder aber eine automatische Parallelisierung von Programmen. Wir konzentrieren uns hier auf die automatische Parallelisierung.

Für viele regelmäßige Programme mit statisch bekannten Ablaufstrukturen, etwa solche mit Matrixrechnungen an zentraler Stelle, ist eine automatische Übersetzung für Parallelrechner inzwischen möglich [95]. Andere Bereiche entziehen sich aber nach wie vor der automatischen Parallelisierung oder zumindest der effizienten parallelen Ausführung. Eine Klasse solcher Programme zeichnet sich durch rekursive Berechnungen aus, deren nicht statisch vorhersehbares, datenabhängiges und unregelmäßiges Ablaufverhalten herkömmliche parallelisierende Übersetzer scheitern läßt. Viele naturwissenschaftliche, mathematische oder grafische Probleme fallen in diesen Bereich.

Methoden und Werkzeuge, die sich automatisch den potentiellen Parallelismus unabhängiger Rekursionsäste in solchen Programmen zunutze machen, erscheinen daher erstrebenswert.

1.2 Stand der Forschung

Dieser Abschnitt gibt nur einen groben Überblick der relevanten Arbeiten in diesem Gebiet. Für eine ausführliche Diskussion sei auf Kapitel 3 verwiesen.

Viele erfolgreiche Ansätze zur Parallelisierung existieren für regelmäßig strukturierte Programme, die z.B. Berechnungen auf dicht besetzten Matrizen durchführen und deren Ablaufverhalten statisch bekannt ist. Dies gilt besonders für explizit parallele Sprachen. In der Umgebung von FORTRAN mit regelmäßigen Feldern und Schleifen, wo Datenabhängigkeitsanalysen greifen, ist auch die automatische Parallelisierung relativ weit fortgeschritten. Für unregelmäßige Probleme zeigen diese Verfahren aber nur unbefriedigende Leistungen, sofern sie sich überhaupt anwenden lassen — das Problem läßt sich statisch nicht aufteilen und die anfallenden Arbeitseinheiten haben sehr unterschiedliche Größen, was zu einer schlechten Leistung dieser Verfahren führt.

Im Gegensatz dazu existieren relativ wenige Arbeiten, die sich mit der Parallelisierung von Programmen mit dynamischen verzeigerten Datenstrukturen oder Rekursionen auseinandersetzen. Hier muß der Programmierer bislang explizit Sprachkonstrukte zur Ausnutzung des Parallelismus einsetzen.

Eine andere Klasse von Arbeiten betreibt die Parallelisierung von rekursiven Teile-und-Herrsche-Algorithmen durch Einführung von Sprachkonstrukten oder ganzen Sprachen oder durch Bereitstellung spezieller Bibliotheken. Diese Ansätze haben zumeist Effizienzprobleme bei unregelmäßiger Struktur des Programmablaufs.

Einige Projekte nutzen zur Laufzeit gewonnene Daten, um spätere Programmläufe zu optimieren, etwa zur Vorhersage von Cacheleistung zur Auswahl von Optimierungen oder für die Analyse des potentiellen Parallelismus von annotierten sequentiellen Programmen.

Meine Arbeit beschäftigt sich mit der automatischen Parallelisierung unregelmäßiger rekursiver Programme. Sie führt spezielle Parallelisierungsstrategien für solche Programme ein und beschreibt Verfahren, die die automatische Auswahl und Parametrisierung von Strategien basierend auf maschinell gewonnenen Laufzeitprofilen eines Programms erlauben. Als Konkretisierung wird ein System zur automatischen Anwendung dieser Verfahren entworfen, realisiert und untersucht.

1.3 Ziel

Die vorliegende Arbeit verfolgt zwei Thesen:

1. *Durch geeignete Parallelisierungsstrategien kann der Parallelismus rekursiver Verzweigungen zur effizienten Beschleunigung des Programmablaufs ausgenutzt werden. Eine solche Parallelisierung kann automatisch durch Modifikation des Programm-Quellcodes vorgenommen werden.*

2. *Die Wahl einer geeigneten Parallelisierungstrategie ist aufgrund von Laufzeitdaten des Programms automatisch möglich. Diese Daten lassen sich durch eine automatische Instrumentierung des Programms gewinnen.*

Daraus ergeben sich für die Arbeit folgende Aufgaben:

Zum einen sind Parallelisierungsstrategien zu entwickeln, die speziell für rekursive Programme eine gute Beschleunigung auf Mehrprozessorrechnern mit gemeinsamem Speicher erreichen. Dabei sollen nicht nur „Spielzeugprogramme“ oder Codefragmente betrachtet

werden, sondern in sich abgeschlossene Anwendungen, die einen möglichst breiten Bereich abdecken.

Zum anderen sind Kenngrößen solcher Programme zu identifizieren, die aus zur Laufzeit gewonnenen Daten abgeleitet werden und einem Programmsystem die Auswahl einer für das jeweilige Problem geeigneten Strategie ermöglichen.

Beide Aufgaben sind durch ein Programmsystem zu automatisieren, d.h. im Idealfall übergibt der Benutzer dem System seinen Quellcode und seine Eingabedaten, das System instrumentiert das Programm zur Datengewinnung, führt es aus, analysiert die gesammelten Informationen, wählt eine Parallelisierungsstrategie aus und parallelisiert das Programm für spätere Programmläufe.

Zur Validierung ist die Leistung des automatisch parallelisierten Programms zu vergleichen mit den Ergebnissen der anderen möglichen Parallelisierungsstrategien. Außerdem soll das Ergebnis einem Vergleich mit handoptimiertem Code standhalten.

Der Vorteil eines solchen Systems liegt auf der Hand: Der Benutzer profitiert von der Beschleunigung, die die parallele Ausführung bietet, ohne daß er viel Zeit und Mühe in eine Parallelisierung von Hand stecken muß oder ihm die Umcodierung in eine spezielle Sprache vorgeschrieben wird — die eventuell in der Praxis nicht einmal ansprechende Laufzeiten erzielt, weil sie rekursive und irreguläre Abläufe nur ungenügend unterstützt.

1.4 Überblick

Im Kapitel 2 werden die Grundlagen von Parallelrechnern, rekursiven Programmen und Irregularität betrachtet. Danach findet in Kapitel 3 die Einordnung in verwandte Arbeiten und der Vergleich mit ihnen und den erzielten Beschleunigungen statt. Basierend darauf definiert Kapitel 4 die Anforderungen und beschreibt den Entwurf von Parallelisierungsstrategien, automatischer Instrumentierung, Parallelisierung und Strategiewahl. Kapitel 5 befaßt sich mit der Realisierung des Systems. Die Darstellung der Ergebnisse in Kapitel 6 umfaßt die verwendeten Benchmarks, die Diskussion der erzielten Beschleunigungen und die Überprüfung der verschiedenen Thesen durch Experimente. Es bietet außerdem eine Untersuchung der Skalierung auf 8 bis 30 Prozessoren und die Auswertung einer Fallstudie mit Praktikumsteilnehmern. Zur Validierung untersucht Kapitel 7 die Leistung des Systems für weitere, bei der Systemkonstruktion nicht berücksichtigte Benchmarks. Das abschließende Kapitel 8 zeigt nochmals die wichtigsten Ergebnisse auf und gibt einen Ausblick auf weitere mögliche Arbeiten. Anhang A stellt einige Systemkomponenten genauer vor, beschreibt Versuche mit Maschinenlernen und visualisiert die Zusammenhänge von Problem und Rekursion an einem Beispiel.

1.5 Beiträge dieser Arbeit

Die an das System gestellten Anforderungen wurden allesamt erfüllt oder übertroffen.

Basis der Aussagen ist eine Benchmarksuite von neun realen rekursiven Programmen, davon vier Benchmarks in der Validierungsmenge, die für Entwurf und Realisierung des Systems völlig unberücksichtigt blieb.

Im Rahmen der Arbeit wurden verschiedene Parallelisierungsstrategien untersucht, die sich speziell auf rekursive Programme beziehen. Die damit erzielten Beschleunigungen liegen je nach Benchmark zwischen dem Faktor 2,8 und 4,0 auf 4 Prozessoren eines Mehrprozessorrechners mit gemeinsamem Speicher und bis zu 33,0 (sic!) auf 30 Prozessoren. Die Strategien

mit ihren jeweiligen Parametern schneiden je nach Benchmark sehr unterschiedlich ab. Diese Messungen spannen den Parameterraum auf, in dem sich die automatische Strategiewahl bewegt. Der Laufzeit-Mehraufwand für Instrumentierung und Erzeugung paralleler Aktivitäten liegt im Bereich von 0,1% bis 4,6%, ist also moderat im Vergleich zur erzielten parallelen Beschleunigung.

Als Existenzbeweis wurde ein System namens REAPAR[1] entwickelt, das in C geschriebene rekursive Programme analysiert, automatisch durch Quellcodetransformationen instrumentiert, Parallelisierungsstrategien durch Heuristiken und Simulation auswählt und die Programme daraufhin parallelisiert. Zur parallelen Ausführung der Rekursionsäste werden Ausführungsfäden (*threads*) verwendet. Abbildung 1.1 visualisiert die Struktur des Systems.

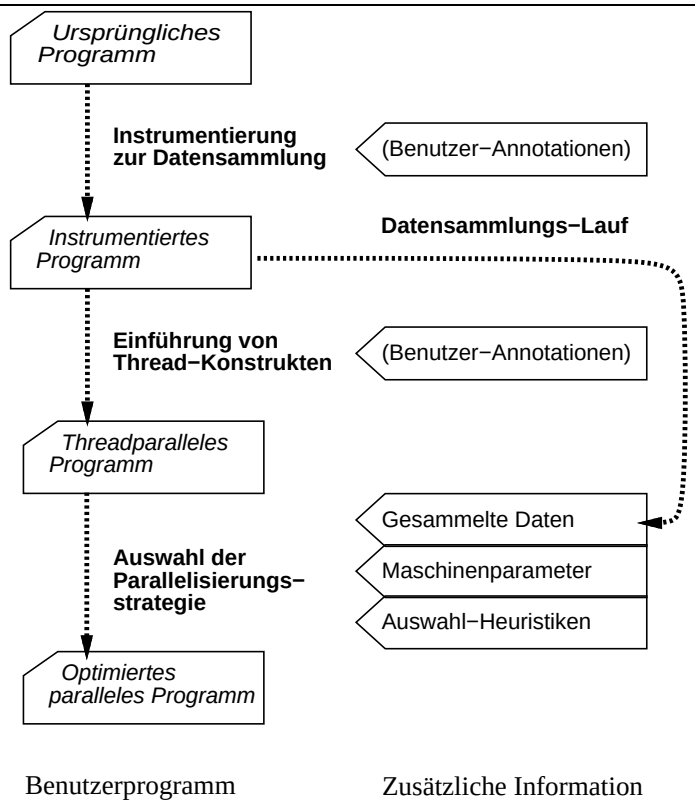


Abbildung 1.1: Aufbau des REAPAR Systems

Es wurden Programmannotationen eingeführt, mit denen der Benutzer z.B. notwendige Synchronisationspunkte im Programm markieren kann. Die mögliche automatische Behandlung solcher Teilprobleme wird aufgezeigt aber nicht realisiert. Zur Threadbehandlung wird die Infrastruktur des Solaris-Betriebssystems verwendet.

Eine Fallstudie zeigt, daß das System Leistungen erzielt, die vergleichbar mit handparallelisiertem Code sind. Die Programme werden in Sekunden automatisch parallelisiert, während eine Handprogrammierung aufwendig und fehleranfällig ist.

REAPAR erreicht dabei die Beschleunigung vergleichbarer Systeme, die auf expliziter Parallelisierung beruhen, oder übertrifft sie sogar.

[1] REcursive programs Automatically PARallelized — reaping the fruits of parallelism

1.6 Zusammenfassung der Ergebnisse

Dieser Abschnitt führt die genannten Ergebnisse der Arbeit quantitativ in Tabellen auf. Er faßt damit die zentralen Resultate von Kapitel 6 und 7 zusammen, ohne auf Details einzugehen. Für weitere Informationen sei auf später verwiesen.

1.6.1 Beschleunigungen im Literaturvergleich

Im Vergleich mit verwandten Systemen wie Cilk und Olden (siehe Kapitel 3) erzielt REAPAR folgende maximale Beschleunigungen für Benchmarkläufe auf vier CPUs:

Benchmark	Barnes Hut	Bitonic Sort *	Eigenvalue	Fractal	Heat *	Knapsack *	Magic *	Power	Queens
Olden	3,0	2,3	—	—	—	—	—	3,8	—
Cilk	3,1	—	—	—	3,9	3,6	—	—	3,9
REAPAR	3,4	3,4	4,0	3,8	4,2	3,1	3,7	3,6	3,9

Die mit „*“ markierten Benchmarks gehören zur Validierungsmenge des Systems, die erst nach Abschluß der Realisierung untersucht wurde. Ihr Abschneiden belegt, daß REAPAR nicht nur auf die ursprünglichen Benchmarks hin optimiert wurde, sondern daß die entwickelten Methoden allgemeingültig sind.

1.6.2 Qualität der automatischen Strategiewahl

Die folgende Tabelle zeigt, welcher Prozentsatz der Beschleunigung der optimalen Strategie durch die automatisch gewählte Parallelisierungsstrategie typischerweise erreicht wird:

Benchmark	Barnes Hut	Bitonic Sort	Eigenvalue	Fractal	Heat	Knapsack	Magic	Power	Queens
Leistung	100%	100%	100%	100%	95%	90%	100%	100%	100%

Bis auf zwei Ausnahmen wählt REAPAR also eine Parallelisierungsstrategie, die auch in der Realität am besten abschneidet. Die besten Strategien sind je nach Problem sehr unterschiedlich, d.h. es gibt keine universelle Strategie.

1.6.3 Beschleunigungen auf mehr Prozessoren

Nach Abschluß der Arbeit ergab sich kurzfristig die Möglichkeit, einige der von REAPAR parallelisierten Programme ohne weitere Anpassung auf Maschinen mit 8, 12 und 30 Prozessoren durchzumessen. Die Beschleunigungen und die Qualität der Strategiewahl sind im Folgenden aufgeführt:

Benchmark	Barnes Hut	Eigenvalue	Fractal	Power	Queens
Beschleunigung 8 CPUs	5,0	8,3	7,3	7,3	4,2
Strategieleistung	100%	100%	73%	—	100%
Beschleunigung 12 CPUs	5,9	16,5	10,0	7,3	3,4
Strategieleistung	100%	100%	50%	—	100%
Beschleunigung 30 CPUs	8,2	33,0	20,0	9,8	2,8
Strategieleistung	11%	100%	100%	—	32%

Das hervorragende Abschneiden von Eigenvalue zeigt, daß grobgranulare Programme perfekt skalieren können. Trotz Problemen wie zu kurzen Programmlaufzeiten (Fractal), zu geringem Parallelitätspotential (Power) und zu feiner Granularität (Queens) ist die Mehrzahl dieser Ergebnisse mehr als zufriedenstellend und unterstreicht die Gültigkeit der Methoden von REAPAR auch für weit höhere Prozessorzahlen als den ursprünglichen vier CPUs.

Kapitel 2

Grundlagen

Dieses Kapitel beschreibt die für die Arbeit relevanten Grundlagen und ihr Umfeld: Parallelrechner zur Leistungssteigerung gegenüber sequentieller Programmverarbeitung, rekursive Programme und ihre Möglichkeiten zur Problemformulierung, und die Unregelmäßigkeit im Ablauf solcher Programme als Herausforderung an die effiziente Parallelisierung.

Allgemein verstehen wir im folgenden unter *Programmen* die in einer Programmiersprache notierten Ausprägungen eines *Algorithmus*. *Prozeduren* sind Gliederungseinheiten eines Programms. Ein *Problem* besteht aus einem Programm und einem Satz *Eingabedaten*.

2.1 Parallelrechner

Die Rechenleistung eines einzelnen Mikroprozessors erhöht sich seit Jahrzehnten um jährlich 50% [29]. Die Supercomputer von gestern stehen heute auf jedem Schreibtisch. Wenn aber eine noch höhere Leistung mit der aktuell verfügbaren Technologie gefordert ist, muß ein Problem auf mehrere Prozessoren verteilt werden.

Ziel der Parallelisierung ist eine möglichst hohe *Beschleunigung* (*speedup*) des Programmablaufs auf n Prozessoren, d.h. das Verhältnis von sequentieller Laufzeit t_{seq} zu paralleler Laufzeit t_{par}. Im Idealfall liegt die Beschleunigung bei n. Eine abgeleitete Maßzahl ist die *Effizienz* $t_{seq}/(t_{par} * n)$. Eine Effizienz nahe 1 gibt an, daß der durch die Parallelisierung hervorgerufene Mehraufwand gering ist.

2.1.1 Architekturen

Um Parallelrechner zu kategorisieren, kann man sie unter verschiedenen Gesichtspunkten betrachten, etwa der Art der Befehlsabarbeitung, der Speicherorganisation oder der Topologie des Verbindungsnetzwerks. Für die vorliegende Arbeit sind besonders die folgenden Unterscheidungen wichtig, wobei der Schwerpunkt auf Rechnern mit gemeinsamem Speicher liegt:

2.1.1.1 SIMD und MIMD

Diese Einteilung unterscheidet nach der Zahl der Instruktions- und Datenströme einer Maschine [34]. Eine sequentielle Maschine wird als SISD (*single instruction single data*) bezeichnet.

SIMD steht für *single instruction multiple data*, also die parallele Ausführung desselben Programmbefehls auf unterschiedlichen Datenelementen. Klassische SIMD-Rechner sind et-

wa die Connection Machine [48] oder die MasPar [10]. In ihnen arbeiten Tausende von relativ schwachen Prozessoren im Gleichschritt den selben Befehlsstrom des zentralen Kontrollprozessors ab, wenden ihn aber auf jeweils prozessoreigene Daten an.

Dieser Architektur erlaubt eine einfache datenparallele Programmierung [49] von regelmäßigen Problemen, deren Daten vektorartig angeordnet werden können, und bei denen sich die Zuordnung von Daten und Prozessoren nicht dynamisch unvorhersagbar ändert.

Heute werden hauptsächlich MIMD-Rechner eingesetzt (*multiple instructions multiple data*), bei denen die Prozessoren unabhängig voneinander verschiedene Teile eines Programms bearbeiten. Die Programmierung ist hier aufwendiger, weil kein klares Modell wie das der datenparallelen Verarbeitung zutrifft und die Synchronisation im Gegensatz zu SIMD-Rechnern explizit vorgenommen werden muß.

Dafür können MIMD-Rechner sehr flexibel eingesetzt werden, z.B. durch Partitionierung der Prozessoren in gleichzeitig von verschiedenen Anwendern nutzbare Bereiche, und sind wegen ihrer Verwendung von Standardprozessoren kostengünstiger. Da es sich bei den Rechenknoten dieser Maschinen um autonome Prozessoren mit eigenem Betriebssystem handelt, kann auch ein einzelner Prozessorknoten selbst nochmals pseudoparallele Aktivitäten ablaufen lassen — Prozesse oder Ausführungsfäden (*threads*, s.u.).

2.1.1.2 Gemeinsamer und verteilter Speicher

Eine andere Art, Parallelrechner zu klassifizieren, ist die Betrachtung ihrer Speicherorganisation, wie Abbildung 2.1 zeigt.

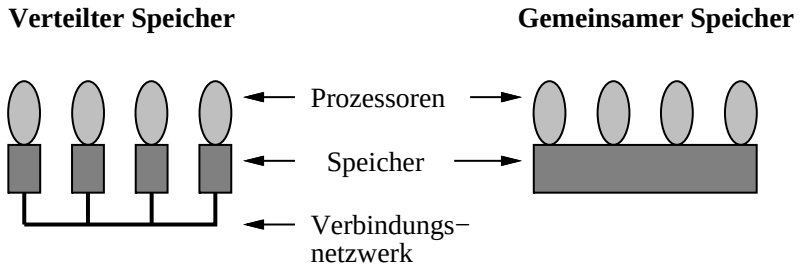


Abbildung 2.1: Unterscheidung von Parallelrechnern anhand der Speicherorganisation

Bei Rechnern mit verteiltem Speicher (*distributed memory*) verfügt jeder Prozessor über seinen lokalen Speicher, auf den er schnell zugreifen kann. Daten, die auf anderen Prozessoren liegen, müssen explizit angefordert und über das Verbindungsnetzwerk des Rechners transportiert werden. Daher werden diese Rechner mit Blick auf ihre Programmierung als nachrichtengekoppelt (*message passing*) bezeichnet. Ihr Vorteil liegt in ihrer großen Skalierbarkeit, die nur durch das Verbindungsnetzwerk begrenzt ist. Dem gegenüber steht die in der Regel aufwendige Programmierung mit explizitem Nachrichtenaustausch und sehr leistungskritischer Verteilung der Daten auf die Prozessoren. Zudem sind für ihre effiziente Nutzung eventuell Maßnahmen zur Verbergung der Kommunikations-Latenzzeiten nötig, wie sie etwa Warschko [93] beschreibt.

Im Gegensatz dazu greifen in Rechnern mit gemeinsamem Speicher (*shared memory*, SMP *Symmetric Multiprocessing*) alle Prozessoren gleichberechtigt auf den selben physikalischen Hauptspeicher zu. Dadurch entfällt die explizite Datenplazierung auf Prozessoren, aber die Skalierbarkeit der Rechner wird durch den gemeinsamen Speicherzugriff beschränkt —

typische Prozessorzahlen liegen im Bereich von 4 bis 128, während Rechner mit verteiltem Speicher in den massiv parallelen Bereich mit mehr als tausend Prozessoren vorstoßen.

In beiden Architekturen verfügen die Prozessoren wie heute üblich über schnelle lokale Zwischenspeicher (*caches*), die sich auch auf die Leistung von Programmen auswirken. Ihr Einfluß wird in dieser Arbeit nicht betrachtet, da er eine grundsätzliche Frage auch bei sequentiellen Rechnern darstellt, extrem rechnerspezifisch ist und sich nur schwer modellieren läßt.

Die vorliegende Arbeit bezieht sich auf MIMD-Rechner mit gemeinsamem Speicher, so daß im folgenden hauptsächlich auf für diese Architektur relevante Informationen eingegangen wird.

2.1.2 Maschinenmodelle

Um parallele Algorithmen theoretisch zu analysieren, wurden verschiedene Modellierungen von Parallelrechnern vorgeschlagen. Die populärsten sind PRAM und LogP. Im PRAM-Modell [35] greifen beliebig viele Prozessoren synchron in Einheitszeit auf einen gemeinsamen Speicher zu.

Das LogP-Modell [25] berücksichtigt zusätzlich die Tatsache, daß Speicherzugriffe je nach Lokalität der Daten verschiedene Kommunikationskosten mit sich bringen (Latenz, Mehraufwand und Bandbreite) und nur eine feste Zahl von Prozessoren vorhanden ist. Außerdem erfolgen die Speicherzugriffe asynchron.

Für die in dieser Arbeit betrachten unregelmäßigen rekursiven Programme bringt keines dieser Modelle einen Nutzen, da das Programmverhalten datenabhängig ist und sich nicht statisch vorhersagen läßt, also keine Optimierung aufgrund der Modelle erlaubt. Allgemeine theoretische Vorhersagen wären für die in der Realität auftretenden Abläufe dieser Programme äußerst komplex, insbesondere wegen der Caches, so daß sich diese Arbeit auf die phänomenologische Klassifizierung des Programmverhaltens beschränkt und darauf basierend mit Heuristiken eine Parallelisierung vornimmt. Die Ergebnisse geben diesem pragmatischen Ansatz recht.

2.1.3 Herausforderungen

Das Hauptproblem von Parallelrechnern ist die eingangs erwähnte Schwierigkeit, ihre Leistung für konkrete Programme voll auszunutzen. Der effizienten parallelen Ausführung stehen mehrere Probleme entgegen:

2.1.3.1 Datenabhängigkeiten

Nur die wenigsten Befehle eines Programms können unabhängig voneinander parallel abgearbeitet werden. Viele Berechnungen beruhen auf Ergebnissen anderer Programmabschnitte, was die Parallelität einschränkt. Ein Programm, dessen Datenabhängigkeiten einen sequentiellen Ablauf vorschreiben, kann nur durch Umcodieren oder Ändern des zugrundeliegenden Algorithmus — wenn überhaupt — parallelisiert werden.

In der Praxis genügt es, wenn die rechenintensiven Teile eines Programms unabhängig voneinander sind, also z.B. die Berechnungen, die 90% der Programmlaufzeit ausmachen, sich in voneinander unabhängige Teile partitionieren und parallel ausführen lassen.

Die vorliegende Arbeit betrachtet nur Programme, bei denen eine solche Datenunabhängigkeit gegeben ist. Dies ist gerechtfertigt, denn das Erkennen von Datenabhängigkeiten

als solches ist Kern vieler weiterer Arbeiten [54], und die automatische Umformung von allgemeinen Programmen in eine Version, die sich besser zur Parallelisierung eignet, ist weder in Theorie noch in Praxis absehbar.

2.1.3.2 Datenlokalität und Caches

Besonders bei Rechnern mit verteiltem Speicher ist es für die Leistung eines Programms essentiell, daß die benötigten Daten lokal auf dem jeweiligen Prozessor vorliegen, da der Zugriff auf nichtlokale Daten um mehrere Größenordnungen langsamer ist. Abhilfe kann die oben genannte Latenzverbergung schaffen.

Rechner mit gemeinsamem Speicher zeigen dieses Problem naturgemäß nicht, wenn man von den erwähnten Cache-Effekten absieht. Diese Effekte werden im REAPAR-System nur phänomenologisch berücksichtig: Das System zielt auf eine effiziente Parallelisierung ab, die definitionsgemäß nur erreicht werden kann, wenn es keine großen negativen Cache-Effekte gibt. Solche Effekte treten z.B. auf, wenn mehrere CPUs an einem Problem arbeiten und dabei auf benachbarte Daten zugreifen, die dann zwischen den Caches der einzelnen CPUs verteilt werden und Zugriffskonflikte verursachen, anstatt lokal im Cache einer CPU zu liegen. Andererseits sind auch überlineare Beschleunigungen möglich, wenn Daten disjunkt zwischen den CPUs verteilt werden und der Datenanteil jeder CPU in ihren Cache paßt, während die Gesamtheit der Daten im Einprozessorfall zu groß für den Cache ist. Eine genauere Diskussion findet sich z.B. bei Anderson et al. [1].

Um nicht noch die optimale Verteilung von Daten auf Prozessoren berücksichtigen zu müssen, die bei unregelmäßigen Programmen mit unvorhersagbaren Kommunikationsmustern äußerst schwierig ist, beschränkt sich diese Arbeit auf Maschinen mit gemeinsamem Speicher. Im Ausblick werden Aspekte der möglichen Verwendung von verteiltem Speicher aufgezeigt.

2.1.3.3 Lastverteilung

Damit ein Parallelrechner effizient ausgenutzt ist, müssen seine Prozessoren möglichst zu jedem Zeitpunkt eine sinnvolle Berechnung durchführen. Eine ungleiche Verteilung der Arbeitslast führt zu brachliegenden Ressourcen und schlechter Beschleunigung.

Zum Thema Lastverteilung (*load balancing, scheduling*) gibt es ein großes Angebot an Literatur. Ziel ist immer die gleichmäßige Verteilung von gegebenen (Teil-) Aufgaben auf die Prozessoren, was durch die allgemein nicht vorhersagbare Laufzeit der Aufgaben erschwert wird.

Die vorliegende Arbeit setzt eine Ebene darüber an: Sie verwendet die Lastverteilung des Betriebssystems und versucht, eine angemessene Anzahl von Arbeitseinheiten geeigneter Größe zur Verfügung zu stellen. Dadurch wird sie unabhängig von der Infrastruktur und erspart sich den hohen Aufwand, den die konkrete Realisierung eines Schedulers mit sich bringt.

2.1.3.4 Granularität

Auf existierenden MIMD-Maschinen kann Parallelität nur effizient genutzt werden, wenn die Größe der parallel ausgeführten Arbeitseinheiten nicht zu gering ist: Die parallele Addition von zehn Zahlen rechtfertigt z.B. nicht den Mehraufwand, eine parallele Aktivität (*thread*) zu initiieren. Eine Parallelisierung lohnt sich also nur ab einer Mindestgröße der Teilprobleme, wobei Teilprobleme zu größeren Problemen zusammengefaßt werden können.

2.1.3.5 Programmierung

Die bisherigen Punkte waren prinzipieller Natur. Für den realen Einsatz von Parallelrechnern stellt sich zudem das Problem der einfachen Programmierung — bisher hat sich noch keine Sprache herausgebildet, die einheitlich auf allen parallelen Architekturen mit guten Leistungen für alle Probleme übersetzbar ist. Bei hohen Leistungsanforderungen wird immer noch sehr maschinennah programmiert, was geringe Portabilität und vor allem Fehleranfälligkeit des Codes und hohen Wartungsaufwand mit sich bringt. Die Energie des Programmierers wird von Maschinendetails und Widrigkeiten wie schwer reproduzierbaren Laufzeitfehlern und Synchronisationsproblemen verzehrt, anstatt die Lösung des eigentlichen Problems voranzutreiben.

Der Idealfall der Programmierung wäre ein Übersetzer, der aus einem gegebenen sequentiellen Programm ein effizientes paralleles Programm erstellt. Für einige Gebiete wie datenparallele reguläre Programme gibt es solche Ansätze, aber eine allgemeine Lösung ist noch nicht in Sicht. Die vorliegende Arbeit bietet Methoden und ein Programmiersystem, um unregelmäßige rekursive Programme automatisch zu parallelisieren und einen möglichst effizienten Ablauf zu erreichen.

2.1.3.6 Fazit

Der effiziente und leicht zu handhabende allgemeine Einsatz von Parallelrechnern liegt noch in weiter Ferne, aber in Teilgebieten gibt es Ansätze, die die Anwendung solcher Rechner wesentlich erleichtern. Die vorliegende Arbeit erweitert diese Forschung im Bereich der rekursiven Programme.

2.2 Threads

Ein zentraler Begriff im praktischen Teil dieser Arbeit ist der des Ausführungsfadens (*thread*). Threads ermöglichen den effizienten Ablauf mehrerer Aktivitäten innerhalb eines einzelnen Prozesses. Im folgenden werden die für die Arbeit relevanten Punkte aufgeführt, ohne einen Anspruch auf die vollständige Beschreibung der Ablaufdetails zu erheben.

Jedes moderne Betriebssystem unterstützt die pseudo-parallele Ausführung mehrerer sogenannter Prozesse. Dazu wird die Rechenzeit durch das Betriebssystem zwischen den verschiedenen Aktivitäten in Zeitscheiben aufgeteilt. Mit einem Prozess ist eine Vielzahl von Verwaltungsinformationen verbunden, etwa eine Identifikation des Benutzers, der den Prozess gestartet hat, die aktuell offenen Dateien, die Größe und Position von Programm- und Datenteilen, eine Priorität, der aktuelle Programmzähler und Stapelzeiger und vieles mehr. Zudem läuft jeder Prozess geschützt vor fremden Zugriffen in einem eigenen Adreßraum ab. Der Wechsel zwischen Prozessen und ihre Erzeugung und Beendigung erfordert daher einen vergleichsweise hohen Aufwand, was die Zahl der sinnvoll einsetzbaren Prozesse in einem System begrenzt.

Andererseits gibt es in Programmen oft Funktionseinheiten, die nur schwach miteinander zusammenhängen, gemeinsame Daten nutzen und unabhängig voneinander ablaufen können oder sogar sollen, z.B. die Interaktionskomponente eines Programms mit grafischer Benutzeroberfläche und seine eigentlichen Rechenprozeduren. Diese Einheiten greifen auch auf gemeinsame Daten zu und wechseln sich oft ab: während des Wartens auf Benutzereingaben oder langsame Festplattenzugriffe können produktive Berechnungen durchgeführt werden.

Allgemein können also Teile einer Anwendung, die auf disjunkten Teilen gemeinsamer Daten arbeiten, unabhängig voneinander ausgeführt werden.

In diesen Fällen bieten sich „leichtgewichtige Prozesse“ an, die im selben Betriebssystemprozeß ablaufen und sich dessen Adreßraum teilen. Dies ist das Konzept der Threads. Das Umschalten, Erzeugen und Beenden von solchen Aktivitäten erfordert wesentlich geringeren Aufwand als die Behandlung vollwertiger Prozesse. Zudem erlaubt die Verwendung eines gemeinsamen Speicherbereichs einen schnellen Datenaustausch. Abbildung 2.2 verdeutlicht diese Zusammenhänge.

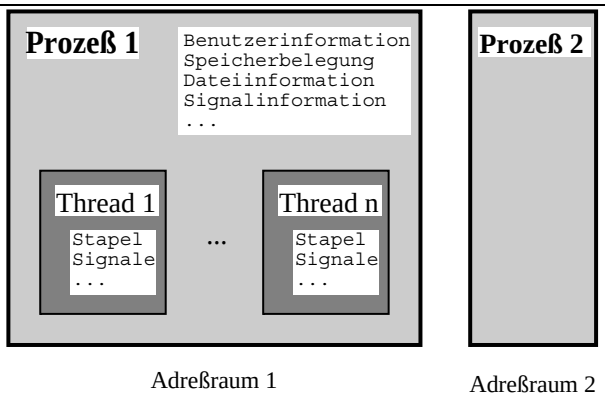


Abbildung 2.2: Betriebssystemprozesse und enthaltene Threads

Zur Abstimmung der Aktivitäten und zur Vermeidung von Konflikten beim Zugriff auf gemeinsam genutzte Daten bietet ein Thread-System auch Operationen für Synchronisation, Warten auf Ereignisse, Sperren kritischer Abschnitte und viele mehr. Es gibt eine POSIX-Norm für die Programmierschnittstelle von Threads, so daß die Portabilität von Thread-Programmen gewährleistet ist (IEEE Standard 1003.1c-1995).

Die Ausnutzung von Parallelrechnern geschieht durch Threads also auf eine natürliche Art und Weise — der pseudo-parallele Ablauf auf einem Einzelprozessor wird durch den real parallelen Ablauf auf mehreren Prozessoren ersetzt. Wenn genügend arbeitende Threads vorhanden sind, kann das volle Parallelitätspotential einer Maschine optimal ausgenutzt werden. Dabei gibt es zwei Grenzfälle, die zu vermeiden sind: Wenn zuwenige Threads existieren, können nicht alle Prozessoren der Maschine ausgelastet werden. Auf der anderen Seite kann es zu übermäßigem Verwaltungsaufwand und entsprechend geringer Effizienz der Parallelisierung kommen, wenn das Programm zuviele Threads verwendet, die nur sehr kleine Teile eines Problems bearbeiten und immer neu erzeugt und beendet werden.

Es ist also kritisch für eine gute Beschleunigung eines Programms, die Zahl der Threads in einem für die Maschine geeigneten Rahmen zu halten.

2.3 Rekursive Programme

Ein weiterer Kernbegriff der vorliegenden Arbeit ist die Rekursion. Unter einer rekursiven Prozedur verstehen wir pragmatisch eine Prozedur f, die sich selbst direkt oder über weitere Prozeduren hinweg aufruft. Formal:

Sei $f\ ruft\ g$ die statische Aufrufrelation, d.h. im Programmcode der Prozedur f steht ein Aufruf der Prozedur g. Nur zur Laufzeit bestimmbare Aufrufe z.B. durch Prozedurzeiger

seien ausgeschlossen. Dann ist f rekursiv, wenn $f\ ruft\ f$ oder $\exists\ h_1 \ldots h_n$: $h_1\ ruft\ h_2 \wedge \ldots \wedge h_{n-1}\ ruft\ h_n \wedge h_1 = f \wedge h_n = f$.

Ebenfalls als rekursiv bezeichnen wollen wir einen Aufruf einer rekursiven Prozedur g durch eine rekursive Prozedur f, auch wenn f danach nicht mehr durch g aufgerufen wird (die Rekursion „bleibt unter sich“).

Das ebenso klassische wie nutzlose[1] Beispiel einer Rekursion ist die Berechnung von $n!$ als $n! = n * (n-1)!$ mit $0! = 1$. Solche einfachen Rekursionen ohne Verzweigungen sind mit theoretischen Mitteln wie dem Aufstellen von Rekurrenzrelationen [23] leicht zu beschreiben. Sie lassen sich auch durch Übersetzer automatisch in eine iterative Version überführen (*tail recursion elimination*).

Im Gegensatz dazu sind im Umfeld dieser Arbeit solche rekursiven Prozeduren interessant, die sich mehr als einmal verzweigen, also mehrere rekursive Aufrufe innerhalb einer Prozedur durchführen. Ihr dynamischer Aufrufgraph ist dann keine lineare Liste mehr, wie bei $n!$, sondern ein Baum. Wenn die Verzweigungen zusätzlich noch in Abhängigkeit von Daten erfolgen, dann greifen theoretische Mittel nur noch zur Abschätzung von besten, mittleren oder schlechtesten Fällen, wenn überhaupt. Zur Beherrschung solcher unregelmäßigen Probleme sind spezielle Techniken notwendig — der Begriff der Unregelmäßigkeit wird später nochmals genauer betrachtet.

Rekursive Programme, die für diese Arbeit interessant sind, finden sich in vielen Bereichen der Physik, der Mathematik, der Wirtschaftssimulation etc. Darunter fallen auch die meisten Programme, die nach dem Teile-und-Herrsche Prinzip vorgehen (*divide and conquer*).

2.3.1 Begriffsbestimmungen

Basierend auf dieser informellen Beschreibung führe ich einige Begriffe im Umfeld der Rekursionsbäume ein, die zentral für die folgende Arbeit sind. Abbildung 2.3 verdeutlicht diese Begriffe. Wenn der Kontext eindeutig ist, wird hier die Prefix „Rekursions-“ zur besseren Lesbarkeit auch weggelassen.

Allgemein sei $Kinder(k)$ die Menge der direkten Kinder (Unterknoten) des Knotens k im Baum, $Vater(k)$ der direkt übergeordnete Knoten von k, $Unterbaum(k)$ der Unterbaum mit Wurzel k, *Knoten(b)* die Menge der Knoten im (Unter)Baum b und $|b|$ die Zahl der Knoten in b, wobei Blätter *Blätter*(b) Knoten mit null Kindern entsprechen.

Aufrufbaum: Während des Programmablaufs baut sich implizit ein Baum der Prozeduraufrufe auf:[2] Die Hauptprozedur bildet die Wurzel und jeder Aufruf einer Prozedur g durch eine Prozedur f stellt einen Knoten für f mit einem Kind-Knoten für g dar. Aufrufe mehrerer Prozeduren entsprechen Verzweigungen im Baum.

Rekursionsbaum: Der durch den Aufruf einer rekursiven Prozedur erzeugte Baum von rekursiven Prozeduraufrufen, also ein Teilbaum des Aufrufbaums eines Programms.

An der Wurzel des Rekursionsbaums steht der initiale Aufruf an eine rekursive Prozedur durch eine nichtrekursive Prozedur. Knoten sind Aufrufe rekursiver Prozeduren — entweder direkte Aufrufe einer Prozedur an sich selbst, oder Aufrufe anderer Prozeduren, die wiederum rekursiv sind. Die Blätter des Baums bilden rekursive Prozeduraufrufe, die ihrerseits dynamisch keine weiteren rekursiven Prozeduren aufrufen, also am

[1] Laut Donald Knuth sollte diese Funktion am besten mittels einer Nachschlage-Tabelle realisiert werden, da sie sehr schnell den maschinendarstellbaren Zahlenbereich verläßt.

[2] Hiervon abgegrenzt sei der Begriff des Aufrufgraphen, der sich statisch auf den Programmcode bezieht und in dem die Beziehung „welche Prozedur ruft welche auf“ dargestellt wird. In diesem Graphen entsprechen rekursive Prozeduren Zyklen.

Pseudocode

```
/* Hauptprogramm */

main() {
  ...
  f(z)          /* Initialer Aufruf    */
  ...
}

f(x) {
  ...
  if (cond) {
    f(x-1)      /* Direkte Rekursion   */
  } else {      /*  Verzweigungsgrad 1 */
    g(x)        /* Aufruf an ebenfalls */
  }             /*  rekursive Prozedur */
  ...
}

g(x) {
  ...
  if (cond) {
    g(x/2)      /* Rekursion mit       */
    g(x/2-1)    /*  Verzweigungsgrad 2 */
  }
  h(x)          /* Keine Rekursion     */
  ...
}

h(x) {
  ...
}
```

Resultierender Aufrufbaum und eingebetteter Rekursionsbaum

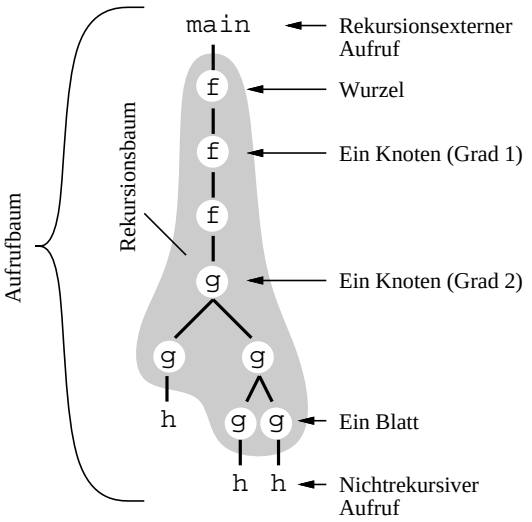


Abbildung 2.3: Beispiel eines Rekursionsbaum mit zugehörigem Pseudocode.

Ende der Rekursion angelangt sind. Eventuell folgende Aufrufe nichtrekursiver Prozeduren schlagen sich nicht im Rekursionsbaum nieder; ihre Ausführung wird dem Blatt zugeordnet. Wenn es mehrere initiale Aufrufe an rekursive Prozeduren gibt, entstehen mehrere Rekursionsbäume.

Im Rekursionsbaum nicht berücksichtigt werden Datenabhängigkeiten. Der Baum der Aufrufreihenfolge $f(x); f(y); use\_results; f(z)$ ist identisch mit dem von $f(x); f(y); f(z)$. In Abschnitt 5.6.3 wird dieses Verhalten nochmals genauer diskutiert.

Unterbaum: Ein Knoten im Rekursionsbaum und alle von ihm rekursiv durchgeführten Prozedurenaufrufe.

Rekursionstiefe: Die Rekursionstiefe $T(f)$ eines konkreten Prozeduraufrufs f entspricht seiner Tiefe im Rekursionsbaum. Der initiale Aufruf an der Wurzel des Rekursionsbaums hat die Tiefe 0. Der Aufruf einer Prozedur g durch eine Prozedur f hat die Tiefe $T(g) = T(f) + 1$. Unter der Tiefe $T(b)$ eines Rekursionsbaums b verstehen wir die maximal erreichte Tiefe aller enthaltenen Prozeduraufrufe.

Verzweigungsgrad: Ein Knoten k im Rekursionsbaum, d.h. der Aufruf einer Prozedur f, hat den Verzweigungsgrad $V(k) = v$, wenn aus ihm v rekursive Aufrufe hervorgehen, d.h. die Prozedur f ruft während ihres Ablaufs v weitere rekursive Prozeduren direkt auf ($|Kinder(f)| = v$). Blätter im Rekursionsbaum sind dementsprechend Knoten mit Verzweigungsgrad 0. Der maximale Verzweigungsgrad $V(b)$ eines Rekursionsbaums b ist das Maximum des Verzweigungsgrads aller enthaltenen Prozeduraufrufe.

Oft wird eine iterative Formulierung eines Programms einer rekursiven vorgezogen, da der rekursive Prozeduraufruf einen etwas höheren Aufwand gegenüber einer Schleife darstellt und Platz auf dem Programmstapel verbraucht. Andererseits ist die rekursive Formulierung eines Problems kompakter und meistens einfacher zu verstehen und zu realisieren — die rekursive Formulierung etwa der schnellen Fouriertransformation ist in drei Zeilen notiert, während das iterative Äquivalent wesentlich länger und komplizierter ist.

2.3.2 Parallelitätspotential

Für die Parallelisierung von Programmen ist noch ein weitere Faktor von nöten: Sofern die rekursiven Aufrufe untereinander keine Datenabhängigkeiten zeigen, können die Verzweigungen eines rekursiven Programms ohne weitere Umformulierung des Problems parallel ausgeführt werden. Genauere Anforderungen dazu werden in Abschnitt 4.3.1.1 aufgeführt.

Rekursionen bieten also ein Parallelitätspotential, das sich — wie diese Arbeit zeigt — sogar automatisch effizient ausnutzen läßt.

2.4 Unregelmäßigkeit

Der Begriff der Unregelmäßigkeit oder Irregularität bezieht sich in dieser Arbeit auf Programme und ihre Rekursionsbäume. Unregelmäßige Probleme sind eine Herausforderung, weil sie sich wegen ihrer ungleich verteilten, unvorhersehbaren Last nicht statisch auf die Prozessoren eines Parallelrechners verteilen lassen, bzw. bei statischer Verteilung eine schlechte Effizienz erbringen.

Wir geben vor der genauen Definition zunächst eine intuitive Abgrenzung unregelmäßiger Probleme durch zwei Kriterien:

Unbalanciertheit der Rekursionsbäume: Ein voll besetzter Baum entspricht weniger der Intuition eines unregelmäßigen Aufbaus als einer, dessen rechte Äste wesentlich mehr Blätter enthalten als die linken. Abbildung 2.4 zeigt eine solche Situation — beim linken Rekursionsbaum sind die beiden Unterbäume des Wurzelknotens perfekt balanciert, während sie beim rechten Baum deutlich unterschiedlich groß sind.

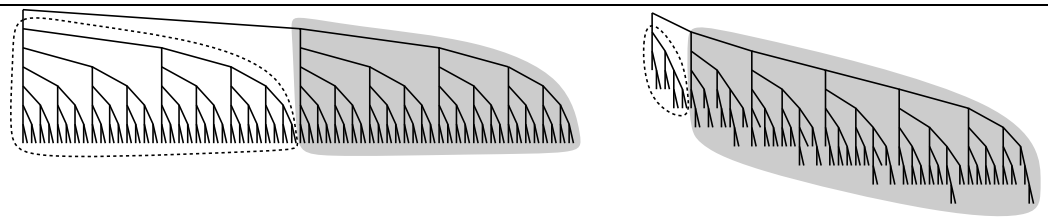


Abbildung 2.4: Balancierter und unbalancierter Rekursionsbaum

Unvorhersagbarkeit der Abläufe: Der Programmablauf und die resultierenden Prozeduraufrufe sind nicht statisch aus dem Programmcode ableitbar, sondern geschehen dynamisch in Abhängigkeit von den Eingabedaten. Bei einer Matrixmultiplikation ist z.B. bei gegebener Problemgröße statisch bekannt, welche Rechenschritte wie oft und in welcher Reihenfolge ablaufen. Eine Galaxiensimulation, deren Rekursionsbaum abhängig von den Simulationsdaten ist, erlaubt eine solche Vorhersage hingegen nicht.

In den Arbeiten von Löwe [61] wird der Begriff der *planbaren Programme* definiert, die sich durch statisch bestimmbare Ablaufmuster (Kommunikation) auszeichnen. Die vorliegende Arbeit betrachtet Programme, die nicht planbar sind.

Vor diesem Hintergrund definieren wir die *Unbalanciertheit* eines Knotens als den Faktor, um den sein größter Unterbaum größer ist als sein kleinster Unterbaum. Die Unbalanciertheit des gesamten Baums läßt sich dann als der Prozentsatz aller Knoten ausdrücken, die höchstens um einen bestimmten Faktor unbalanciert sind, z.B. „60% aller Knoten sind maximal um den Faktor 2 unbalanciert“.

Formal: $|k|$ bezeichne abkürzend die Zahl der Knoten in $Unterbaum(k)$. Das *größte Kind* $Kmax(k)$ eines Knotens k ist dann definiert durch $\forall c_i \in Kinder(k) : |Kmax(k)| \geq |c_i|$. Analog ist das *kleinste Kind* $Kmin(k)$ definiert durch $\forall c_i \in Kinder(k) : |Kmin(k)| \leq |c_i|$.

Einen (Unter)Baum mit Wurzelknoten k nennen wir *u-unbalanciert*, wenn seine direkten Kinder sich um maximal den Faktor u in der Größe unterscheiden: $u = |Kmax(k)| \,/\, |Kmin(k)|$, d.h. das größte Kind ist u mal so groß wie das kleinste. Ein perfekt balancierter Baum ist also 1-unbalanciert, da alle Unterknoten gleich groß sind.

In Verfeinerung nennen wir einen ganzen (Unter)Baum b $(p; u)$-unbalanciert, wenn ein Anteil von p seiner Knoten höchstens u-unbalanciert sind, d.h. $\left|\{k : k \in Knoten(b) - Blätter(b) \wedge u \leq |Kmax(k)| * |Kmin(k)|\}\right| = p * |Knoten(b)|$. In diese Analyse geht die Balance eines Blatts nicht ein, da sie immer 1 ist. Bei festem u läßt sich das zugehörige p durch Analyse des Baums leicht bestimmen. Umgekehrt kann man für festes p den Parameter u solange variieren, bis p erreicht ist — die Aussage ist dann „wieviel Unbalanciertheit u muß man tolerieren, um p% der Knoten abzudecken?“.

Der linke Beispielsbaum aus Abbildung 2.4 ist also (1; 1)-unbalanciert, da alle seine Knoten gleichgroße Unterbäume haben. Sein Wurzelknoten ist perfekt 1-unbalanciert. Der rechte Beispielsbaum ist hingegen (0,61; 1)-unbalanciert, da nur 54 seiner 88 Knoten gleichgroße Unterbäume haben. Sein Wurzelknoten ist 9,35-unbalanciert, weil seine beiden Unterbäume so verschieden groß sind. Die Balancierung für u=1,5 ist (1; 1,5) bzw. (0,70; 1,5), die Werte für u=3, d.h. ein maximaler Unterschied um den Faktor 3 bei der Unterbaumgröße, sind (1; 3,0) bzw. (0,97; 3,0). Es gibt also auch im geometrischen Baum nur wenige Knoten, deren Unterbaumgrößen sich um mehr als 3 unterscheiden — für eine Parallelisierung ist es natürlich schon schlecht, wenn genau der Wurzelknoten ein großes Ungleichgewicht aufweist und alle darauffolgenden Unterbäume unbalanciert sind. Beim regelmäßigen Baum sind bereits bei Balanciertheit 1,0 mindestens 66% der Unterbäume abgedeckt, beim unregelmäßigen Baum muß dazu hingegen eine Unbalanciertheit von 1,4 in Kauf genommen werden.

Bei der Beschreibung der Benchmarks im Kapitel 6 werden jeweils die Unbalanciertheit des Wurzelknotens, die (p_1; 1,0) / (p_2; 1,5) / (p_3; 3,0) Werte und der (0,66; u) Wert angegeben. Die Wurzelbalance zeigt, wie ungleichmäßig die Knotenverteilung der obersten Baumebene ist — eine Wurzelbalance nahe 1 läßt auf einen gleichmäßig verteilten Gesamtbaum schließen, während ein großer Wert bedeutet, daß zumindest die obersten Ebenen des Baums sehr große Unterschiede aufweisen (die Struktur der Unterbäume kann ihrerseits besser verteilt sein). Die drei Werte $p_{1...3}$ geben an, wieviele Knoten im Baum gleichgroße / um 1,5 unterschiedliche / um den Faktor 3,0 unterschiedliche Unterbäume haben. Große Werte von p_i bedeuten eine gleichmäßige Verteilung der Blätter im Baum, geringe Werte eine Ungleichverteilung. Üblicherweise ist $p_1 < p_2 < p_3$ (mit $\leq$ statt $<$ gilt diese Relation immer). Ähnliche niedrige Werte für $p_{1...3}$ deuten darauf hin, daß große Teile des Baums eine Unbalance von mehr als 3 haben. Der (0,66; u)-Wert schließlich zeigt, wie stark das Unbalanciertheitskriterium u erhöht werden muß, damit es von mindestens 66% aller Knoten erfüllt wird.

Kapitel 3

Verwandte Arbeiten

Dieses Kapitel bietet einen Überblick verwandter Arbeiten, die in mindestens einem der Bereiche Parallelisierung, Rekursion, Irregularität oder Verwendung von Laufzeitdaten relevant sind, und vergleicht sie mit der vorliegenden Arbeit.

3.1 Explizit parallele Sprachen

Es gibt viele erfolgreiche Ansätze zur expliziten Parallelisierung regelmäßig strukturierter Programme, die z.B. Berechnungen auf dicht besetzten Matrizen durchführen, und deren Ablaufverhalten statisch bekannt ist. Sie basieren auf entsprechenden Programmiersprachen, etwa Modula-2* [77], und erzielen für geeignete Probleme sehr gute Beschleunigungen. Auf solche rein datenparallelen Sprachen soll im weiteren nicht mehr eingegangen werden, da sie weder Rekursion noch Irregularität effizient unterstützen können. Hierzu zählt für die Belange dieser Arbeit auch High Performance FORTRAN (HPF [36, 58]), mit seinem datenparallelen `forall` Konstrukt und spezifischer Unterstützung für die Datenverteilung auf Rechnern mit verteiltem Speicher.

NESL [11, 44, 76] von G.Blelloch an der CMU verfolgt einen verschachtelt datenparallelen Ansatz in einer ML-ähnlichen Sprache mit expliziten Konstrukten für Parallelausführung und Sequenz-Datentypen. Rekursionen sind leicht möglich, wobei innerhalb eines Rekursionsschritts datenparallel gearbeitet wird. Auch irreguläre Probleme lassen sich dabei ausdrücken. Ursprünglich für Vektor- und SIMD-Rechner gedacht, gibt es inzwischen Portierungen für Rechner mit verteiltem Speicher, die allerdings mit Leistungsproblemen zu kämpfen haben (Beschleunigung von 3-5 auf 48 CPUs!). Das System besteht aus Übersetzer, interpretierter Zwischensprache für eine abstrakte Vektormaschine und rechnerspezifischer Laufzeitbibliothek. Neuere Arbeiten zu NESL [11] betrachten weniger die Laufzeit als vielmehr die Arbeit („work“, Zahl der Operationen) und die Tiefe („depth“, längste Abhängigkeits-Kette im Programm), passend zu rekursiven Programmen. Ein Übergang zu spezialisierten sequentiellen Prozeduren ab einem bestimmten Unterteilungsgrad des Systems soll die Leistung erhöhen.

Das von J.Feldman et al. am ICSI in Berkeley entwickelte **pSather** [33, 88] erweitert die objektorientierte Eiffel-ähnliche Sprache Sather um parallele Konstrukte, die Threads erzeugen. Synchronisation und Daten/Objektplazierung sind explizit und maschinenabhängig, Rekursion und irreguläre Programme werden unterstützt. Einen maschinenunabhängigeren Ansatz mit nachfolgender Optimierung von Prozeßzahl, Kommunikation und Synchronisationselimination verfolgt **parSather** [62].

Durch seine Threadkonstrukte kann **Java** [2] an dieser Stelle ebenfalls als explizit parallele portable Sprache verstanden werden. Verschiedene Arbeiten verfolgen auch die weitere Integration von Threads in Sprachen, z.B. **Multithreaded C** von J.Thornley et al. am CalTech [15], das sequentielle Programme um Annotationen zur Parallelisierung von Schleifen und Programmblöcken sowie Synchronisationskonstrukte ergänzt. Eine ähnliche Richtung verfolgt das ältere **CC++** [20].

P.B. Hansens **SuperPascal** [42] zielt auf effiziente parallele Rekursion ab, aber betont Aspekte auf niedriger Systemebene wie die Speicherallokation — der Autor zitiert sich weitgehend selbst und es gibt keine Leistungsvergleiche. Das Allokationsproblem stellt sich im Rahmen der vorliegenden Arbeit überhaupt nicht.

OPAL [87, 94] von A.Wehrenpfennig aus Dresden bietet parallele objektorientierte Programmierung für Transputer. Rekursive Programme sind möglich. Die Maschinenausnutzung durch Erzeugung genügend vieler Objekte (z.B. bis zu einer bestimmten Rekursionstiefe) bleibt dem Programmierer überlassen, das Laufzeitsystem verteilt danach die Last. Tuning-Komponenten ermöglichen Messungen verschiedener Laufzeit-Parameter, aufgrund derer der Benutzer das Programm verbessern kann.

Zudem gibt es mehrere Spezialsprachen, die für die Parallelisierung von Anwendungen eines bestimmten zugeschnitten sind, wie **CuPit** [52] von L.Prechelt aus Karlsruhe, das gezielt die parallele Behandlung neuronaler Netze und ihren Ablauf u.a. auf SIMD-Rechnern unterstützt.

All diesen Sprachen ist gemeinsam, daß sie im Gegensatz zu REAPAR keine automatische Parallelisierung durchführen, sondern nur eine mehr oder weniger hohe Abstraktion von der Maschine bieten. Die geeignete Problemformulierung und Parallelisierung bleibt dem Benutzer überlassen.

3.2 Automatische Parallelisierung

Im Bereich des „Scientific Computing“ ist die automatische Parallelisierung relativ weit fortgeschritten, besonders in der oben erwähnten Umgebung von FORTRAN mit regelmäßigen Feldern und Schleifen, wo auch Datenabhängigkeitsanalysen greifen [4, 27, 95]. Diese Ansätze zeigen aber Probleme mit dynamischen irregulären Programmen, da die notwendigen Konstruktionen nur unzureichend unterstützt werden — z.B. werden dynamisch zur Aufrufzeit geschachtelte `forall`s in HPF rein sequentiell abgearbeitet.

„Implicitly Parallel C“ — **IPC** [47, 75] — von J.Hicks et al. aus dem Motorola Cambridge Research Center erweitert C um die `sync` Storage Class, die eine nur einmal schreibbare Variable kennzeichnet, ähnlich der Semantik funktionaler Sprachen. Daraus kann der Übersetzer die Datenabhängigkeiten ableiten und Instrumentierungen einfügen, die potentielle Parallelität zur Laufzeit erkennen. Der gesamte Sprachumfang von C einschließlich Funktionszeigern soll in der Endversion unterstützt werden. Zielarchitektur sind Thread-parallele Maschinen mit gemeinsamen Speicher; die Arbeit zielt auf eine zweifache Beschleunigung auf vier CPUs ab. Nach der verfügbaren Literatur wurden nur Parallelismus-Profile aufgestellt, aber das System erzeugt noch keinen Code für parallele Maschinen, weshalb keine Messungen vorliegen. Der Aufwand für Threaderzeugung und Datenzugriffe wird noch nicht berücksichtigt. Die beiden Beispielprogramme sind sehr klein — eine Behandlung realer Programme ist für zukünftige Arbeiten vorgesehen, die aber noch nicht vorliegen.

REAPAR bietet hingegen ein real existierendes System, das kein Umschreiben der Programme erfordert und automatisch parallelisiert.

An der Indiana University wird von A.Bik et al. die Parallisierung rekursiver Java-Programme mit dem **JAVAR**-System untersucht [7]. Dabei soll der Übersetzer ein annotiertes Programm in explizit threadparallelen Code transformieren, wobei auch Schleifen parallelisiert werden sollen. Die Grundidee ist verwandt mit der vorliegenden Arbeit, hat aber weitergehende Einschränkungen bei der Parallelisierung (statische Zuteilung ohne Unterstützung irregulärer Programme, nur eine Parallisierungsstrategie für Rekursionen). Zudem wirft Java als objektorientierte Sprache weitere Probleme wie den erst zur Laufzeit bestimmten Methodenaufruf auf, die nicht überzeugend geklärt werden. Die Literatur läßt offen, welche Teile des Projekts reine Vorhaben und welche schon realisiert sind. Die Beispiele beschränken sich auf kleine Programme wie Sortieralgorithmen und die aktuelle Version des Systems [8] benötigt noch eine explizite Kennzeichnung potentieller Parallelität durch den Benutzer, bietet also keinerlei Automatismus.

Ein neuer Ansatz von M.Rinard und P.Diniz [82] benutzt **Kommutativitätsanalyse**, um in einer Untermenge von C++ geschriebene objektorientierte Programme automatisch zu parallelisieren. Dabei werden Objekte identifiziert, deren Ausführung ohne Auswirkung auf das Rechenergebnis vertauscht werden kann, und Aufrufe dieser Objekte später parallel ausgeführt. Ein spezielles Laufzeitsystem erzeugt nur dann eine parallele Aktivität, wenn ein Prozessor frei ist. Der Übersetzer transformiert C++ Quellcode in explizit parallelen C++ Code und parallelisiert sowohl Rekursionen als auch Schleifen. Messungen auf dem DASH Spezialrechner mit gemeinsamem Speicher zeigen gute Beschleunigungen, die für höhere Prozessorzahlen gegenüber einer handoptimierten Version zurückfallen. Basis der Aussagen sind drei von den Autoren in C++ umgeschriebene größere Benchmarks mit dynamischem irregulärem Ablauf, unter anderem auch Barnes Hut. Die Beschleunigung für diesen Benchmark ist auf vier Prozessoren 15% besser als die von REAPAR. Im Vergleich zur vorliegenden Arbeit verfolgt dieses System einen analytischen Ansatz ohne Laufzeitrückkopplung, stellt höhere Anforderungen an das zu parallelisierende Programm (Umschreiben in reines C++ etc.) und untersucht nur wenige Benchmarks auf einer Spezialmaschine — Ergebnisse für SGI-Rechner werden erwähnt, aber nicht beschrieben. Eine breitere Basis Benchmarks, die nicht von den Autoren selbst komplett neu geschrieben wurden, würde die vielversprechenden Ergebnisse des Ansatzes besser untermauern. Für nicht objektorientierte Sprachen sind zudem die Voraussetzungen für die erforderliche Programmanalyse nicht gegeben.

3.3 Parallelisierung „dynamischer Programme“

Einige Arbeiten beschäftigen sich mit der Unterstützung „dynamischer Programme“, also Programmen mit verzeigerten Datenstrukturen, bei denen herkömmliche Parallelisierungsansätze versagen. Die meisten dieser Systeme eignen sich natürlicherweise für rekursive Programme; viele bieten auch Mechanismen zur Behandlung irregulärer Strukturen. Der größte Teil der Benchmarks, die in der vorliegenden Arbeit verwendet werden, ist aus Programmen dieser Systeme abgeleitet.

Cilk [12, 13, 57, 91] von R.Blumofe et al. am MIT bietet eine Multithread-Ausführung für C-Programme, in denen der Parallelismus durch Konstrukte wie `spawn` und `sync` explizit gemacht wurde. Es besteht aus einen Präprozessor mit Typüberprüfung und einem sehr aufwendigen Laufzeitsystem, dessen Scheduler beweisbare Garantien bezüglich der Ausführungszeit gibt. Dabei wird noch die Laufzeit und die bei optimaler Parallelität minimal erreichbare Laufzeit gemessen. Die Laufzeitparameter werden aber nicht vom System für weitere Optimierungen ausgewertet, und es gibt keine verschiedenen Parallelisierungsstrate-

gien. Der Scheduler ist *lazy*, d.h. nicht alle durch Konstrukte eingeführte Parallelität wird wirklich genützt, was für feingranulare Programme Vorteile bringt. Rekursive und irreguläre Programme werden unterstützt, und ein `abort` Konstrukt erlaubt den asynchronen Abbruch einer parallelen Berechnung, z.B. für Suchalgorithmen. Im Laufe des Projekts wurde Cilk auf diverse Maschinen (Connection Machine CM-5, Netze von Arbeitsplatzrechnern, Sun Enterprise Server etc.) portiert. Die zahlreichen Beispielprogramme beinhalten kleine Tests wie Fibonacci und Teile-und-Herrsche-Programme, aber auch Galaxiensimulation und sogar ein erfolgreiches Schachprogramm. Weitgehende Messungen belegen eine gute Beschleunigung für die meisten Benchmarks.

Cilk kommt von den verwandten Arbeiten dem REAPAR-System von der Handhabung und den neueren unterstützten Maschinen her am nächsten. Im Gegensatz zur vorliegenden Arbeit verlangt es aber eine explizite Parallelisierung durch den Programmierer, führt keine automatischen Messungen zur Optimierung späterer Läufe durch und basiert auf einem komplexen Laufzeitsystem mit Eingriffen in Stapelverwaltung und Betriebssystemabläufe, während REAPAR die vorhandene Infrastruktur ausnutzt. Fünf der REAPAR-Benchmarks sind von Cilk-Benchmarks abgeleitet und erreichen oder übertreffen die publizierten Leistungen von Cilk.

Das System **Olden** [16, 83, 17] von A.Rogers et al. aus Princeton verwendet eine Untermenge von C mit Annotationen zur Parallelisierung von Programmen mit dynamischen Datenstrukturen, d.h. Zeiger-basierten Daten und rekursiven Prozeduren zu ihrer Behandlung. Der Programmierer kennzeichnet parallelisierbare Aufrufe durch `futurecall` und erzwingt ein Warten auf das Ergebnis durch eine `touch`-Operation. Außerdem wird dem verteilten Speicher der Zielmaschinen durch explizite Allokation von Daten auf bestimmten Prozessoren Rechnung getragen. Die Berechnung migriert dann Threads zu den gerade verwendeten Daten oder das Laufzeitsystem entscheidet aufgrund von Heuristiken und Hinweisen des Programmierers, daß ein Software-Caching effizienter ist. Die Lokalität ist dabei nur so gut, wie die Allokation der Daten auf den CPUs durch den Programmierer. Die Autoren geben eine Sammlung von elf teilweise umfangreichen Benchmarks und genaue Messungen der Beschleunigung auf iPSC/860 und CM-5 an. Für die Zukunft ist eine Automatisierung der Parallelisierung basierend auf älteren Arbeiten von L.Hendren [45, 46] geplant.

Im Gegensatz zu REAPAR ist bei Olden bereits durch die Zielmaschinen mit verteiltem Speicher ein hoher Aufwand des Programmierers nötig, um gute Leistungen zu erzielen. Olden parallelisiert zur Zeit in keiner Weise automatisch, und die Qualität einer automatischen Parallelisierung im Vergleich zur aktuellen aggressiven Handparallelisierung bleibt abzuwarten. Die drei von Olden angepaßten Benchmarks liefern bis auf eine Ausnahme unter REAPAR deutlich höhere Beschleunigungen.

In Berkeley entwickelten D.Culler et al. das **Split-C** System [24, 64], das eine Mischung von Datenparallelität, gemeinsamem Speicher und Nachrichtenübermittlung für Maschinen mit verteiltem Speicher bietet. Diese Maschinenklasse mit ihren Kommunikationsoperationen bedingt ganz andere Schwerpunkte des Systems: Der Programmierer spezifiziert genau die Datenverteilung und kann Optimierungen der Kommunikation z.B. durch Vorabanforderungen oder Massen-Datentransfers vornehmen — der Übersetzer führt selbst kaum Optimierungen durch. Zur Parallelisierung wurde das zugrundeliegende C um Zugriffs- und Datenlayout-Primitive sowie globale Zeiger ergänzt. Ein einfaches Kostenmodell unterscheidet zwischen billigen lokalen Zugriffen und teuren globalen. Irreguläre Probleme werden behandelt, aber Rekursion wird nicht unterstützt. Anwendung findet das System in der Lehre. Außerdem zeigen größere Beispielprogramme auf Maschinen wie Cray T3D und CM-5 gute Beschleunigungen.

Von den Systemen dieses Abschnitts ist Split-C das maschinennähste, das auch Änderungen am Algorithmus der Programme erfordert und keinerlei Automatisierung bietet. Leistungsvergleiche mit REAPAR sind schwer, weil Split-C die Leistung bevorzugt in MFLOPs angibt und Beschleunigungen nur als Kurven zeigt, die im relevanten Bereich von 4-8 Prozessoren nicht genau abzulesen aber offensichtlich sublinear sind. Die angegebene Effizienz von 75% liegt deutlich unter der von REAPAR.

3.4 Spezielle Ansätze für Teile-und-Herrsche

Das Teile-und-Herrsche Prinzip [53, 84], bei dem ein Problem rekursiv in Unterprobleme zerlegt wird, die gelöst und dann zu einer Lösung des Gesamtproblems zusammengesetzt werden, bietet von sich aus ein großes Potential an Parallelität. Viele Programme lassen sich in diese Klasse einordnen. Dementsprechend gibt es Systeme, die sich genau auf diese Algorithmen spezialisieren, um paralleles Programmieren zu erleichtern.

Eine Reihe von Arbeiten geht Teile-und-Herrsche-Algorithmen im Parallelen aus einer funktionalen Perspektive an — stellvertretend sei hier **Divacon** von G.Mou [71, 72] aus Yale erwähnt, das eine Algebra für solche Algorithmen aufstellt und sie in einer speziellen Sprache formuliert. Rekursion ist damit natürlicherweise behandelt, aber der Ansatz scheitert für datenabhängige Unterteilungen, d.h. nur die „üblichen“ regulären Beispiele wie FFT, Matrixmultiplikation und Dreiecksmatrix-Lösung lassen sich behandeln. Eine ähnliche Richtung schlagen auch S.Nishimura et al. [74] oder der Powerlist-Ansatz von J.Misra [69] ein, die einen breiten theoretischen Rahmen schaffen, aber ebenfalls nur für reguläre Rekursionen geeignet sind und keine realistischen Anwendungen vorzeigen können. Vorschläge wie die von T.Axford [3] haben ähnliche Probleme.

Ein weiteres mögliches Vorgehen ist die Verwendung von Bibliotheken für rekursive und potentiell irreguläre Teile-und-Herrsche-Verfahren. **Beeblebrox** von A.Piper [79, 78] etwa bietet C++ Klassen, in die der Programmierer explizit die Programmphasen einbettet.

Einen flexibleren Ansatz versprechen die **Frames** [56] aus Paderborn, bei denen Experten einen parallelen Rahmen schreiben, in denen dann Anwender ihren sequentiellen Code einhängen. Rahmen können miteinander kombiniert werden. Die mögliche Parametrisierung umfaßt ebenfalls Lastbalancierungsalgorithmen. Das Projekt ist noch nicht sehr weit fortgeschritten. Ähnliche Ideen stellen für funktionale Programme Darlington et al. [26] vor.

APRIL [30, 31] von T.Erlebach an der TU München führt Teile-und-Herrsche Programme auf Rechnern mit verteiltem Speicher aus. Die Sprache ist eine Pascal-Untermenge mit Parallelitätskonstrukten und erlaubt pro Programm nur die Parallelisierung einer einzigen rekursiven Prozedur mit dem festen Verzweigungsgrad zwei. Ein Konstrukt ermöglicht die Abfrage, ob noch Prozessoren verfügbar sind. Anwendungsbeispiele umfassen u.a. eine unoptimierte Fraktalberechnung, die gute Beschleunigungen erzielt, und Layoutsynthese.

Die in diesem Abschnitt beschriebenen Arbeiten haben im Vergleich zu REAPAR einen eingeengteren Anwendungsbereich, verlangen eine völlige Neuprogrammierung des Problems in Spezialsprachen oder Bibliotheken und zeigen Schwächen für irreguläre Rekursionen.

3.5 Parallelisierung existierender Programme und Infrastruktur

Weiterhin existieren viele Arbeiten zur expliziten Parallelisierung existierender Programme von Hand, so z.B. die **Splash**-Benchmarks für hierarchische Berechnungen aus Stanford

[85, 86] für Rechner mit gemeinsamem Speicher, die J.Singh und A.Gupta untersuchen, oder Moleküldynamik-Parallelisierungen von T.Clark und R.v.Hanxleden [21, 22] für verteilten Speicher. Diese Programme sind irregulär und zum Teil auch rekursiv, aber die verwendeten Techniken sind auf das jeweilige Programm abgestimmt und nicht automatisiert.

In diesen Bereich fallen ebenfalls Systeme, die die Parallelisierung irregulärer Probleme für den Programmierer unterstützen, z.B. die **CHAOS** Bibliotheken (ehemals PARTI) von J.Saltz et al. [55, 73] und Parallelisierungsuntersuchungen von S.Chakrabarti und K.Yelick aus Berkeley [18, 19] für Rechner mit verteiltem Speicher. Die hier verwendeten Verfahren beziehen sich auf die Probleme auf solchen Rechnern und sind für Maschinen mit gemeinsamem Speicher uninteressant.

Eine ganze Reihe von Systemen wie **Linda** von D.Gelernter aus Yale [38] oder der **Problem Heap** von J.Staunstrup et al. [70] stellen einen Rahmen für explizite Parallelität und ihre Koordination zur Verfügung. Sie verwalten und verteilen die anfallenden Arbeitseinheiten automatisch und unterstützen den nötigen Datenaustausch und Synchronisationen. Im Vergleich zu REAPAR sind sie auf der Ebene der Infrastruktur angesiedelt, da sie keine automatische Unterstützung der Problemformulierung und der Transformation in parallelen Code bieten. Abschnitt 4.3.1.2 ordnet solche Ansätze relativ zur Laufzeitumgebung von REAPAR ein.

REAPAR läßt sich mit diesen Systemen nicht direkt vergleichen, da eine explizite Handprogrammierung von Spezialproblemen nichts mit einer automatischen Parallelisierung rekursiver Programme gemein hat. Interessant ist hingegen der Leistungsvergleich: Einer der Benchmarks der vorliegenden Arbeit (Eigenvalue) wurde von S.Chakrabarti übernommen und liefert sowohl bei diesem als auch bei bei REAPAR perfekte Beschleunigungen auf 4 bis 30 Prozessoren, die per definitionem optimal sind.

3.6 Einbeziehen von Laufzeitdaten

Einige Arbeiten verwenden zur Laufzeit gewonnene Meßergebnisse für weitergehende Optimierungen. Das **P3T** System von T.Fahringer [32] aus Wien mißt z.B. eine Reihe von Benchmarks für jede Zielarchitektur durch, um die Cacheleistung später übersetzter Programme zu erhöhen.

Ein anderer Ansatz von N.Reimer [81] paßt die Zahl der verwendeten Prozessoren dynamisch an die Anforderungen eines irregulären FORTRAN-Programms an, um bestimmte Effizienz- oder Beschleunigungskriterien zu erfüllen, wobei das Programm zuvor von Hand instrumentiert wird.

Das objektorientierte **Self** von U.Hölzle et al. [50] aus Santa Barbara optimiert den dynamischen Methodenaufruf durch Laufzeitinformationen.

Der Fokus dieser Systeme ist ein anderer als REAPAR — es geht nicht um die automatische Parallelisierung, sondern die Optimierung von Teilaspekten einer manuell vorgegebenen Parallelisierung oder im Falle von Self nur um sequentielle Effizienzsteigerung.

3.7 Parallele funktionale und logische Sprachen

Funktionale Sprachen bieten sich auf den ersten Blick ideal für eine automatische Parallelisierung an [92], sind doch die Funktionsaufrufe definitionsgemäß unabhängig und frei von Nebeneffekten. In der Praxis ist es aber problematisch, eine gute Leistung zu erzielen: Die

Granularität eines Funktionsaufrufs kann nicht a priori eingeschätzt werden, d.h. die potentielle Parallelität ist im allgemeinen viel zu feinkörnig für eine effiziente parallele Ausführung [14] — als Lösung werden genau die Methoden vorgeschlagen, die auch REAPAR benutzt, nämlich Laufzeitprofile und ggf. Annotationen durch den Programmierer. Bei *lazy* Sprachen kommt hinzu, daß zur Parallelisierung Berechnungen spekulativ durchgeführt werden müssen, von denen noch nicht feststeht, ob ihre Ergebnisse überhaupt benötigt werden. Ein Überblick paralleler funktionaler Sprachen findet sich z.B. bei S.Martins [67].

Es gibt eine Vielzahl von Arbeiten, die sich mit der Parallelisierung funktionaler Programme beschäftigen, etwa R.Loogen aus Aachen [59], aber die Auswirkung auf reale Anwendungen ist recht gering, da imperative Sprachen wie FORTRAN und C dominieren.

Prolog und andere logische Sprachen bieten ebenfalls ein Parallelitätspotential, da ihre Suchbäume parallel abgearbeitet werden können. Ein Beispiel ist **PARLOG** [39] von S.Gregory aus Bristol. Die Techniken, die hierbei zum Einsatz kommen [40], sind aber nicht mit REAPAR vergleichbar, da REAPAR im Gegensatz zur spekulativen Berechnung bei Prolog auf die Ergebnisse jeder einzelnen Teilberechnung angewiesen ist. Zudem bringt gerade bei der Baumsuche eine gute sequentielle Abarbeitung, die große Teile des Suchbaums frühzeitig beschneidet, wesentlich höhere Beschleunigungen als eine Parallelisierung sie bieten kann. Für die Verbreitung in der Praxis gilt dasselbe wie für funktionale Sprachen.[1]

3.8 Weitere Arbeiten

Die Arbeiten von Löwe [63, 60] aus Karlsruhe ermöglichen die Analyse von datenparallelen PRAM-Programmen und ihre optimierende Transformation in **LogP**-Programme für Maschinen mit verteiltem Speicher. Dazu muß die Kommunikationsstruktur des Programms für eine feste Eingabegröße unabhängig von den Eingabedaten sein (sog. „planbare Programme“), also genau das Gegenteil der in der vorliegenden Arbeit behandelten irregulären Programme. Die Laufzeit des Optimierungsalgorithmus ist für reale größere Programme sehr hoch — typische Beispielprogramme sind FFT und Präfixsumme.

3.9 Vergleich der erzielten Beschleunigungen

In Vorwegnahme der Ergebnisse in Kapitel 6 und 7 zeigt Tabelle 3.1 die Ergebnisse von Olden [16] und Cilk [91] gegenüber den Resultaten der vorliegenden Arbeit. Zusätzlich liefert der Eigenvalue-Benchmark [19] sowohl in der Literatur als auch bei REAPAR prefekte Beschleunigungen. Für eine genaue Diskussion der Tabelle sei auf Abschnitt 6.2.1.5 verwiesen. Die Leistung anderer Systeme ist wegen Verschiedenheit der Infrastruktur und der verwendeten Benchmarks nicht direkt mit REAPAR zu vergleichen, aber Beschleunigungen von bis zu 4 auf 4 Prozessoren sprechen für sich.

[1] Die Verbreitung eines Systems spricht nicht für oder gegen seine Qualität, aber bei den betrachteten Systemen werden durch die Rahmendefinitionen bereits viele Probleme ausgeschlossen, die bei imperativen Sprachen auftreten und dort gelöst werden müssen. Außerdem ist die sequentielle Leistung von z.B. C bereits bedeutend höher als die fast aller funktionalen Sprachen.

Tabelle 3.1: Vergleich der mit REAPAR erzielten Beschleunigungen mit den veröffentlichten Werten für dieselben Benchmarks unter Olden und Cilk auf 4 Prozessoren. Die mit (*) markierten Benchmarks sind Teil der Validierungsmenge, die erst nach Abschluß der Entwicklungsarbeiten an REAPAR hinzugenommen wurde. Cilk gibt für den Magic-Benchmark keine Beschleunigungen an.

Benchmark-name	Beschleunigung		
	Olden	Cilk	REAPAR
Barnes Hut	3,00	3,14	**3,38**
Bitonic Sort (*)	2,29	—	**3,36**
Eigenvalue	—	—	**4,00**
Fractal	—	—	**3,78**
Heat (*)	—	3,90	**4,15**
Knapsack (*)	—	**3,60**	3,11
Magic (*)	—	?,?	**3,67**
Power	**3,81**	—	3,63
Queens	—	**3,90**	3,89

3.10 Einordnung und Vergleich

Tabelle 3.2 gibt einen Überblick der verwandten Arbeiten. Dabei werden folgende Abkürzungen verwendet:

Allgemein:
- \+ Gut / voll unterstützt,
- ∘ Vorhanden / lückenhaft,
- – Nicht vorhanden

Autom. Parallelisierung:
- \+ Vollautomatisch,
- **A** Annotationen,
- **E** Explizite Konstrukte

Art der Parallelität:
- **D** Datenparallel (SIMD),
- **P** Prozesse,
- **T** Threads

Zielarchitektur:
- **G** Gemeinsamer Speicher,
- **V** Verteilter Speicher

Systemaufbau:
- **B** Bibliothek,
- **L** Laufzeitsystem,
- **Q** Quellcode-Transformation,
- **Ü** Übersetzer

Zusammenfassend läßt sich sagen, daß auf dem Bereich der automatischen Parallelisierung irregulärer rekursiver Probleme kein System existiert, das die bei Entwurf und Implementierung von REAPAR gestellten Anforderungen erfüllt. Die Leistungen von REAPAR erreichen oder übertreffen zudem die vergleichbarer Systeme.

Tabelle 3.2: Einordnung der verwandten Arbeiten

Kriterium \ Arbeit	Rekursiv	Irregulär	Automatische Parallelisierung	Laufzeitfeedback	Parallelität	Zielarchitektur	Systemaufbau	Sprache	Auswertungen	Beispiele	Bemerkungen
Vorliegende	+	+	A	+	T	G	B,Q	C	+	+	
NESL	+	+	E	−	D	V	B,L,Ü	ML	+	∘	[a]
pSather	+	+	E	−	T	G,V	Ü	Sather	−	−	
OPAL	+	∘	E	∘	P	V	L,Ü	OOP	+	∘	
IPC	+	∘	A	∘	T	G	L,Ü	C	∘	∘	[b]
Javar	+	∘	A	−	T	G	Q	Java	∘	−	
Kommutativ	+	+	+[c]	−	P	G	L,Q,Ü	C++	+	∘	
Cilk	+	+	E	−	T,P	G,V	L,Q	C	∘	+	[d]
Olden	+	+	E[e]	∘[f]	T	V	L,Ü	C	+	+	
Split-C	−	+	E[g]	−	P/D[h]	V	Ü	C	+	+	
Divacon	+	−	E	−	D	SIMD	Ü	Eigene	∘	−	[i]
Beeblebrox	+	∘	E	−	P	V	B,L	C++	∘	∘	
APRIL	+	∘	E	−	P	V	B,Ü	Pascal	∘	∘	
Infrastruktur	+	+	−	−	T,P	V	B,L	div.	+	+	[j]
Löwe	−	−	−	−	P	V	Ü	PRAM	−	−	

[a] Verschachtelte Datenparallelität — innerhalb einer Rekursionsebene wird datenparallel gerechnet

[b] Ziel ist die Parallelisierung existierender Programme, Beschleunigung von 2 auf 4 CPUs angestrebt

[c] Ein Neu- oder Umschreiben der Programme erscheint nötig, um die Systemvoraussetzungen zu erfüllen

[d] Großes System, viele reale Benchmarks

[e] Explizite Prozessorzuweisung bei Datenallokation nötig, Migration der Arbeit zu den Daten

[f] Eigener Profiler als Systemkomponente für Benutzer

[g] Explizit bis hin zu Datenlayout und optimierten Kommunikationsoperationen z.B. für große Datenmengen und zur Überlappung von Kommunikation und Berechnung

[h] Mischung aus Datenparallel und Nachrichtenvermittlung, SPMD, nur ein Thread pro CPU

[i] Nur für datenunabhängige Kommunikationsmuster geeignet

[j] Diverse Bibliotheken und Laufzeitsysteme wie CHAOS

Kapitel 4

Anforderungen und Entwurf

Dieses Kapitel führt die Anforderungen an ein System auf, das den in der Einleitung gestellten Aufgaben gerecht werden soll, und diskutiert die Annahmen und Rahmenbedingungen der Arbeit. Basierend darauf wird der Entwurf des Systems genauer vorgestellt.

4.1 Anforderungen

Wie in der Einführung in Abschnitt 1.3 dargestellt, verfolgt diese Arbeit zwei Thesen:

1. *Rekursive datenunabhängige Verzweigungen lassen sich durch geeignete Parallelisierungsstrategien automatisch effizient parallel ausführen.*

2. *Automatisch gewonnene Lauzeitprofile ermöglichen die automatische Auswahl einer solchen Strategie.*

Unter Berücksichtigung der vorangegangenen Bemerkungen ergeben sich folgende Anforderungen an ein System, das die Thesen dieser Arbeit bestätigen soll:

A1 Es sollen allgemein anwendbare Verfahren zur Parallelisierung rekursiver Programme angegeben werden.

A2 Die Parallelisierung durch das System muß möglichst automatisch ablaufen. Für Bereiche wie Datenabhängigkeitsanalysen, die bereits in anderen Arbeiten abgedeckt wurden, können Benutzerannotationen verwendet werden.

A3 Da die Existenz einer einzigen Paralleliserungsstrategie, die für alle Probleme optimal ist, unwahrscheinlich erscheint, sind Strategien zu entwickeln, die jeweils für ein Spektrum von Problemen gute Beschleunigungen erlauben.

A4 Die Wahl einer geeigneten Parallelisierungsstrategie und eventueller Strategieparameter muß automatisch erfolgen.

A5 Basis der Strategiewahl sollen Messungen früherer Programmläufe sein, aus denen automatisch strategierelevante Kenngrößen des Problems abgeleitet werden — eine Strategiewahl basierend auf reiner statischer Quellcodeanalyse wäre vorzuziehen, ist aber für irreguläre Programme definitionsgemäß nicht möglich.

A6 Die Messung muß automatisch erfolgen. Eventuell dafür notwendige Änderungen am Programmcode sind durch das System automatisch einzubringen.

A7 Die durch das System erreichte Beschleunigung der Programme soll vergleichbar mit der Beschleunigung durch Handparallelisierung und -optimierung sein, maximal um einen Faktor von zwei schlechter.

A8 Als Beispielsprogramme sollen in sich geschlossene Anwendungen aus verschiedenen Gebieten verwendet werden, die einen breiten Bereich abdecken — in Abgrenzung zu Codefragmenten oder Kleinstprogrammen, aus denen leider in vielen Arbeiten die Gesamtheit der Beispiele besteht.

A9 Zur Validierung der Gesamtleistung des Systems ist eine Reihe von während der Arbeit nicht betrachteten Programmen hinzuzuziehen. Dies würde eine eventuelle Optimierung auf genau die verwendeten Benchmarks hin bloßlegen.

4.2 Annahmen

Um den Nachweis der Thesen dieser Arbeit zu führen, muß das System an den Stellen automatisch arbeiten, wo die Beiträge der Arbeit liegen: Automatische Parallelisierung und Strategiewahl sowie automatische Generierung der dafür erforderlichen Zusatzinformationen. In anderen Bereichen existieren bereits andere Arbeiten, deren Ergebnisse als „black box“ in das System integriert werden könnten. Eine existierende Datenflußanalyse ließe sich z.B. verwenden, um Stellen im Programmcode zu identifizieren, an denen rekursiv berechnete Daten verwendet werden, und die nötigen Synchronisatisoperationen einzufügen.

Neben diesen Infrastruktur-Annahmen gibt es auch grundlegende Annahmen mit Auswirkung auf den Entwurf:

Grundannahme des Meß-Ansatzes ist, daß ähnliche Datensätze ein ähnliches Programmverhalten zur Folge haben. Ein Einsatzszenario wäre etwa die Simulation von 20 Kugelsternhaufen, wobei das System den ersten Programmlauf mißt und darauf basierend die Parallelisierung für die restlichen 19 Läufe vornimmt. Ohne diese Ähnlichkeitsannahme ist jeglicher Versuch, Vorhersagen über das Programmverhalten zu treffen, zum Scheitern verurteilt. Bei Eingaben mit völlig anderen Charakteristiken muß ein erneuter Meßlauf vorgenommen werden.[1]

Wie bereits erwähnt, wird vorausgesetzt, daß die zu parallelisierenden rekursiven Prozeduren überhaupt parallelisierbar sind, also keine störenden Datenabhängigkeiten enthalten. Diese Annahme kann wegfallen, wenn eine mächtige Datenabhängigkeitsanalyse zur Verfügung steht, die ihrerseits bereits Gegenstand intensiver Forschung z.B. bei L.Hendren ist [54] und nicht direkt zu meiner eigentlichen Arbeit beitragen würde.

Die zu parallelisierenden rekursiven Prozeduren müssen statisch aus dem Programmcode ablesbar sein, d.h. Funktionszeiger werden nicht unterstützt. Ohne diese Annahme wäre eine dynamische Analyse des Aufrufverhaltens nötig, die ebenfalls eine eigenständige wissenschaftliche Arbeit darstellt und keinen Beitrag zu den eigentlichen Kernbereichen der vorliegenden Arbeit darstellen würde. Die Implikationen für objektorientierte Sprachen werden im Ausblick erläutert.

Von Maschinendetails wie der Auswirkung von Caches wird angenommen, daß die Unterschiede ihrer Auswirkung für verschiedene Strategien gering ist. Für eine Feinabstimmung der Leistung wären Cache-Überlegungen vorteilhaft, aber auch dieser Bereich ist Thema eigener Arbeiten (ein Überblick findet sich in [51]), und die real erreichten Beschleunigungen

[1] Die Auswertung in Abschnitt 6.2.5 zeigt, daß gute Strategien eines Problems in der Realität auch für einen großen Bereich anderer Probleme gute Leistungen erzielen.

vom bis zu n-fachen auf n Prozessoren sind selbst ohne soche Feinabstimmung bereits nahe dem Optimum.

Die Annahme bezüglich der Leistungsfähigkeit von Threads ist, daß der betriebssystemeigene Scheduler von sich aus eine gute Parallelisierung des Ablaufs erreicht, wenn er nur genügend Arbeitseinheiten geeigneter Größe erhält. Auch dieser Annahme gibt die Realität recht.

4.3 Entwurf

Der Entwurf behandelt alle Überlegungen, die zur automatischen Parallelisierung rekursiver Programme und zur automatischen Wahl einer geeigneten Parallelisierungsstrategie nötig sind. Zum einen sind grundlegende Strategien zur Parallelisierung aufgrund der speziellen Eigenschaften rekursiver Programme zu entwickeln. Unter diesen Strategien soll dann automatisch die gewählt werden, die die höchste parallele Beschleunigung verspricht. Dazu müssen Laufzeitgrößen des untersuchten Programms definiert und automatisch gewonnen werden. Schließlich ist das Programm automatisch so zu parallelisieren, daß die gewählte Strategie beim Programmablauf zur Wirkung kommt.

Globale Entwurfsentscheidungen waren die folgenden:

- Minimalistischer Ansatz mit Ausnutzung vorhandener Infrastruktur, sofern diese sich leicht integrieren läßt. Oben genannte Gründe führten zu dieser Forderung.
- Wahl von Mehrprozessorrechnern mit gemeinsamem Speicher als Zielarchitektur, weil sie weite Anwendung finden, die zusätzlichen Probleme der Datenlokalität und des Nachrichtenaustauschs vermeiden und für die Realisierung des Systems direkt am Institut verfügbar waren.
- Verwendung von C als Zielsprache, da diese Sprache sehr weit verbreitet und hinreichend maschinenunabhängig ist, die „*real world*"-Ausrichtung des Systems unterstreicht und viele als Benchmark verwendbare Programme bereits in C vorliegen.
- Nutzung von Threads zur Parallelisierung von Programmen. Auf den benutzten Mehrprozessorrechnern existieren ausgereifte Threadbibliotheken für C, und auf jedem modernen Betriebssystem gibt es Threads, was die Portabilität erhöht.
- Aufbau des Systems, angelehnt an die UNIX-Philosophie, als eine Reihe von Werkzeugen, die sich zusammenschalten lassen, um den gewünschten Zweck zu erreichen. Dies unterstützt zusätzlich die automatische Durchführung von Experimenten.

Nach diesen globalen Entscheidungen betrachten wir nun die wichtigsten Teilaspekte, die das System abzudecken hat.

4.3.1 Parallelisierungsstrategien für rekursive Programme

Wie bereits im Grundlagenkapitel erwähnt, bieten rekursive Programme mit sich verzweigendem Rekursionsbaum ein hohes Potential an Parallelität. Im folgenden werden Kriterien für die Parallelisierbarkeit solcher Programme vorgestellt und darauf basierend Verfahren zu ihrer Parallelisierung angegeben.

4.3.1.1 Parallelisierbarkeit

Sofern die rekursiven Aufrufe untereinander keine Datenabhängigkeiten zeigen, können sie ohne weitere Umformulierung des Problems parallel ausgeführt werden. Die Parallelisierung geschieht dabei durch die gezielte Erzeugung neuer paralleler Aktivitäten (*threads*) anstelle eines sequentiellen rekursiven Aufrufs. Abbildung 4.1 verdeutlicht dieses Vorgehen — hier wurde der „linke“ rekursive Aufruf durch den ursprünglichen Thread weitergeführt, während für alle anderen Aufrufe jeweils ein neuer Thread erzeugt wurde.

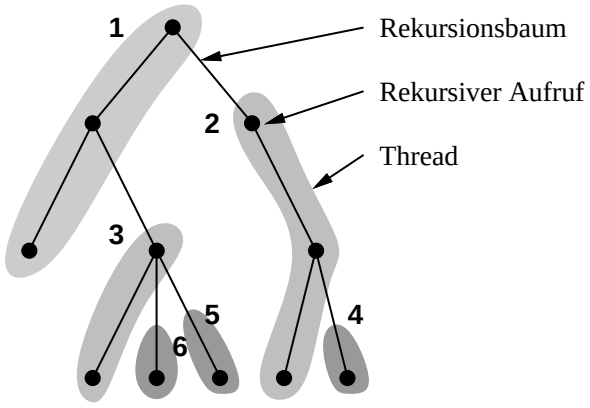


Abbildung 4.1: Parallelisierung eines rekursiven Programms durch Einsatz von Threads. Die grau hinterlegten Threads im Rekursionsbaum laufen parallel zueinander ab.

Ein Programm ist also parallelisierbar, wenn die Unteraufrufe einer rekursiven Prozedur nicht schreibend auf gemeinsame Daten zugreifen, d.h. unabhängige Teile des Problems bearbeiten, und die Ergebnisse voriger Unteraufrufe nicht für weitere Aufrufe benötigt werden. Eine Mitgabe von Parametern durch die aufrufende Prozedur und eine Rückgabe von Werten an sie behindert hingegen eine Parallelisierung nicht. Dabei ist es unerheblich, ob die Prozeduraufrufe textuell nacheinander im Programm stehen oder im Rahmen einer Schleife durchgeführt werden. Die folgende Klassifizierung verdeutlicht diese Zusammenhänge:

Sequentielle Abhängigkeit: Jeder rekursive Aufruf innerhalb der Prozedur $f(x)$ ist vom Ergebnis des vorhergehenden Aufrufs abhängig.

```
f(x):  for (i=1; i<10; i++) {a = a + f(a); ... }
```

Der Rekursionsbaum zeigt also Verzweigungen, bietet aber wegen der Datenabhängigkeiten kein Parallelitätspotential.

Globale Schreibkonflikte: Die rekursiven Aufrufe greifen schreibend auf globale Variablen zu.

```
f(x):  for (i=1; i<10; i++) {global = global + f(a); ... }
```

Auch hier ist eine Parallelisierung ausgeschlossen, da die gleichzeitigen Schreibzugriffe auf globale Variablen beim parallelen Ablauf zu falschen Ergebnissen führen. Selbst wenn man die Schreibzugriffe in einem kritischen Abschnitt kapseln würde, kann das Programm immer noch fehlerhaft parallel laufen, falls die genaue Reihenfolge der rekursiven Aufrufe sich auf das Ergebnis auswirkt (was im Beispiel nicht der Fall ist).

Eltern-Kind-Abhängigkeit: Die Rückgabewerte der rekursiven Aufrufe werden in der aufrufenden Prozedur verwendet, aber die Parameter der einzelnen Aufrufe sind unabhängig davon.

```
f(x):  for (i=1; i<10; i++) {local = local + f(b[i]); ... }
```

In diesem Fall können die Kind-Aufrufe parallel erfolgen. Die aufrufende Prozedur muß auf die Beendigung aller aufgerufenen Prozeduren warten.

Unabhängigkeit: Die rekursiven Aufrufe arbeiten auf sich nicht überschneidenden Daten, entweder eigenen lokalen Daten oder unabhängigen Bereichen globaler Daten. Es werden keine Daten an die aufrufende Prozedur zurückgegeben.

```
f(x):  if (cond) {f(x/2); f(x/2+1);} else {local work}
```

Hier können sämtliche Aufrufe parallel erfolgen. Die aufrufende Prozedur kann sofort nach Abspaltung der Kind-Aufrufe wieder zurückkehren, während die Kind-Aufrufe noch parallel ablaufen.

Die letzten beiden Fälle bieten das geforderte Parallelitätspotential. Ob ein rekursiver Aufruf tatsächlich als Thread gestartet wird oder als normaler Prozeduraufruf abläuft, wird zur Laufzeit durch eine der unten beschriebenen Strategien entschieden.

Bei der Eltern-Kind-Abhängigkeit muß sichergestellt werden, daß vor Beendigung der aufrufenden Prozedur (d.h. an dem Ort, wo die Rückgabewerte der Kind-Aufrufe verwendet werden) die Aktivität der Kinder abgeschlossen ist. Dieser Synchronisationspunkt kann ggf. automatisch aufgrund rein syntaktischer Analyse des Quellcodes vor jedem Rückkehrpunkt der Prozedur gesetzt werden. Wenn die Rückgabewerte bereits früher benötigt werden, muß eine geeignete Stelle für das Warten auf die Kinder durch Datenabhängigkeitsanalyse ermittelt oder durch Benutzerannotation gekennzeichnet werden.

Die Möglichkeit, daß eine Prozedur bereits vor Beendigung der von ihr erzeugten parallel ablaufenden Unteraufrufe zurückzukehrt, bringt konzeptionelle Schwierigkeiten mit sich: Zum einen muß diese Möglichkeit ebenfalls automatisch erkannt oder annotiert werden. Zum anderen stellt sich das zusätzliche Problem, das Ende des initialen rekursiven Aufrufs zu erkennen — da nach Rückkehr eines Prozeduraufrufs die durch ihn initiierten Threads weiterlaufen können, müßte global über alle Threads Buch geführt werden, um das Ende des letzten Threads feststellen zu können.

Daher wird in dieser Arbeit generell der Ansatz verfolgt, vor Prozedurende auf das Ende aller von der Prozedur gestarteten Threads zu warten.

4.3.1.2 Parallelisierungsstrategien

Eine Parallelisierungsstrategie im Sinne dieser Arbeit ist ein Prädikat, das am Ort eines rekursiven Aufrufs bestimmt, ob der Aufruf sequentiell oder mittels eines neuen Threads auszuführen ist. Dieses Prädikat kann mit Parametern versehen sein. Die Zuteilung der erzeugten Threads auf die Prozessoren des Parallelrechners bleibt, wie im Grundlagenkapitel erwähnt, dem Betriebssystem vorbehalten.

Merkmal einer guten Strategie ist, daß sie genügend Threads erzeugt, um die Maschine auszulasten, aber nicht so viele oder so kleine, daß der Mehraufwand der Threadverwaltung die Effizienz beeinträchtigt.

Aus den Anforderungen ergibt sich die Frage nach allgemein verwendbaren Strategien, die für alle rekursiven Programme gleichermaßen anwendbar sind. Erstrebenswert wäre

eine einzige Strategie, die für sämtliche rekursiven Probleme gute Beschleunigungen erzielt. Leider sind die Eigenschaften der Programme, z.B. die Granularität der Berechnungen, so unterschiedlich, daß sich in der Praxis keine solche Strategie angeben läßt. Daher muß die Auswahl einer Strategie und geeigneter Parameter für ein gegebenes Problem durch eine automatischen Strategiewahl erfolgen, die in Abschnitt 4.3.2 behandelt wird.

Eine Klasse von Parallelisierungsstrategien sind die *Allgemeinen Strategien*. Sie verwenden keinerlei programmspezifisches Wissen und wirken überall im Rekursionsbaum gleich. Die Idee der allgemeinen Strategien ist, die Zahl der Threads zu überwachen und gegebenenfalls neue Threads zu erzeugen, wenn deren aktuelle Anzahl unter einen bestimmten Wert fällt. Dieser Wert ist gleichzeitig der Parameter der Strategie. Um eine Skalierbarkeit mit der Maschinengröße zu erreichen, wird er mit der Zahl der Prozessoren multipliziert. Eine mögliche allgemeine Strategie wäre z.B. „erzeuge möglichst fünf Threads für jeden Prozessor", was auf vier Prozessoren insgesamt 20 Threads erzeugen würde. Andere allgemeine Strategien, die keine Parameter benötigen, wären etwa „erzeuge bei jeder Gelegenheit einen Thread" oder „erzeuge niemals einen Thread".

Alle parametrisierten allgemeinen Strategien erfordern, daß die Zahl der aktuell aktiven Threads und die der bisher schon erzeugten Threads aufgezeichnet wird. Dafür ist es nötig, kritische Abschnitte im Programm einzuführen, um Schreibkonflikte des Threadzählers zu vermeiden. Das ruft einen Mehraufwand hervor, der die Effizienz bei sehr feinkörnigen Problemen mindern kann.

Eine andere Strategieklasse ergibt sich aus der Betrachtung des Rekursionsbaums: Es wäre für die Parallelisierung von Vorteil, wenn nur oben im Rekursionsbaum Threads gestartet würden, weil die unteren Baumebenen hauptsächlich aus Blättern und kleinen Teilbäumen bestehen, die die Erzeugung einer eigenen Aktivität kaum lohnen. Dies ist die Idee der *Tiefenstrategie*. Sie erzeugt nur bis zu einer gegebenen Baumebene Threads. Diese Baumebene ist gleichzeitig der Strategieparameter.

Für diese Strategie muß die aktuelle Rekursionstiefe ermittelt werden, was durch automatische Einführung eines zusätzlichen Tiefenparameters für alle rekursiven Prozeduren geschehen kann. Der damit verbundene Mehraufwand an Laufzeit ist vernachlässigbar.

Schließlich lassen sich die obigen Strategien im Zusammenhang betrachten, was zu den *Kombinierten Strategien* führt. Eine kombinierte Strategie wäre z.B. „erzeuge fünf Threads pro Prozessor, aber nur bis zur Rekursionstiefe sieben".

Die folgende Auflistung beschreibt die Strategien und ihre Hauptanwendungsgebiete im einzelnen. Dabei werden zusätzlich prägnante englische Abkürzungen eingeführt, die im weiteren Verwendung finden:

Allgemeine Strategien — Die Entscheidung über die Threaderzeugung wird anhand der aktuellen Zahl von Threads bzw. der bisher erzeugten Zahl von Threads gefällt. Alle Parameter N werden mit der Zahl der Prozessoren P multipliziert. Typische Werte für N bewegen sich im Bereich $1 \ldots 20$.

First N: Nur die ersten $N * P$ Möglichkeiten, einen Thread zu erzeugen, werden wahrgenommen. Das entspricht einer Parallelisierung auf den oberen Baumebenen unabhängig vom Verzweigungsgrad. Anwendungsgebiet sind ausgewogenere Rekursionsbäume, bei denen die ersten Aufrufe bereits genügend Parallelitätspotential bieten.

Keep N: Erzeuge einen neuen Thread, wenn die aktuelle Zahl aktiver Threads kleiner als $N * P$ ist. Die Zahl aktiver Threads wird hierbei erhöht, wenn eine Aktivität

gestartet wird, und erniedrigt, wenn die startende Prozedur die beendete Aktivität „einsammelt“. Diese Strategie verteilt die Arbeit über den Rekursionsbaum — freiwerdende Threads können dort aktiv werden, wo noch Arbeit vorhanden ist.

Active N: Wie Keep N, aber die Zahl aktiver Threads ist anders definiert: Der Threadzähler wird direkt nach Abschluß der parallelen Aktivität erniedrigt, d.h. neue Threads können schon erzeugt werden, bevor die Elternprozedur überhaupt die Ergebnisse ihrer Kinder aufgelesen hat. Dadurch wird im Vergleich zu Keep N eine höhere Zahl von Threads ermöglicht, aber die Zahl der unaufgesammelten Threads kann stark zunehmen.

Always: Starte anstelle jedes rekursiven Aufrufs einen Thread (entspricht Keep ∞) — nützlich für grobkörnige Probleme mit kleinem Rekursionsbaum.

Never: Niemals einen Thread starten (Keep 0), was für Probleme sinnvoll sein kann, bei denen die Threaderzeugungskosten höher sind als die eigentliche Rechenarbeit.

Neverever: Dies ist keine eigentliche Strategie, sondern bezeichnet im folgenden den sequentiellen Ablauf völlig ohne Threadeinführung und ohne kritische Abschnitte. Durch einen Vergleich mit Neverever läßt sich der durch Threads hervorgerufene Mehraufwand bestimmen.

Tiefenstrategie (Depth D): Erzeuge genau in den oberen D Ebenen des Rekursionsbaums Threads (Ebene $0 \ldots D-1$). Diese Strategie vermeidet die Erzeugung von Threads für unrentabel kleine Unterbäume im unteren Bereich des Rekursionsbaums, die besser sequentiell berechnet werden. Da hierbei kein Threadzähler benötigt wird, entfällt der Mehraufwand für kritische Abschnitte im Gegensatz zu den parametrisierten allgemeinen Strategien.

Typische Werte für D liegen im Bereich $1 \ldots 10$, aber tiefe Rekursionsbäume können durchaus Tiefen von 50 erfordern.

Im Gegensatz zur First N Strategie bezieht Depth D bei zunehmendem Parameter D ganze Ebenen des Rekursionsbaums unabhängig von der Anzahl bereits erzeugter Threads mit ein. First N zählt hingegen einzelne rekursive Aufrufe.

Kombinierte Strategien (*Combined Strategies*): Gleichzeitige Verwendung von allgemeiner und Tiefenstrategie. Die allgemeine Strategie begrenzt die Zahl der Threads, und die Tiefenstrategie vermeidet die Parallelisierung kleiner Unterbäume. Allerdings sind aufgrund der allgemeinen Strategie wieder kritische Abschnitte im Programm nötig.

Keep N until Depth D: Erzeuge Threads, wenn die aktuelle Tiefe kleiner ist als D und es weniger als $N * P$ Threads gibt.

Active N until Depth D: Entsprechend, aber mit der Threadzahl-Semantik von Active N.

Wie sich in Abschnitt 6.2.11 herausstellt, bringt die Kombination von Strategien allerdings nicht die erhofften Vorteile, weswegen im weiteren nicht immer auf sie eingegangen wird.

Diese Strategien decken alle rekursiven Programme ohne Datenabhängigkeiten ab. Weitergehende Strategien, die auf problemspezifischer Information wie der Laufzeit von Teilbäumen je nach Eingabegröße aufsetzen, sind denkbar, aber sehr problemspezifisch und von genauer Information seitens des Programmierers abhängig. Sie werden daher nur im Ausblick der Arbeit betrachtet.

Es ist noch anzumerken, daß viele manuelle Parallelisierungen auf der Idee einer globalen Warteschlange mit Teilaufträgen (*task queue*) und einer Menge von Arbeitsprozessen (*worker pool*) bestehen. Eine Variation dieses Ansatzes ist der `problem heap` von Staunstrup et al. [70]. Die Prozesse holen sich jeweils einen Teilauftrag ab, bearbeiten ihn und fordern dann den nächsten Teilauftrag an. Dieses Modell paßt nicht auf das hier betrachtete Problem: Es setzt auf zu niedriger Ebene an — die konkrete Zuordnung von von REAPAR erzeugten Threads auf die Prozessoren des Systems erfolgt durch das Betriebssytem aufgrund von genau solchen Warteschlangen, aber die Entscheidung über die eigentliche Erzeugung einer Aktivität geschieht durch REAPAR eine Ebene darüber. Eine Wiederverwendung von Threads lohnt sich wegen der effizienten Zwischenspeicherung bei Solaris laut Handbuch ausdrücklich nicht [90].

Zum anderen ist es in diesem Modell nicht trivial, die Reihenfolge der Ausführung zu kontrollieren, damit nicht z.B. ein Knoten weiter berechnet wird, bevor alle seine Unterbäume abgeschlossen sind. Solche Garantien bezüglich der Eltern-Kind-Reihenfolge sind hingegen bei der von REAPAR verwendeten Methode trivialerweise gegeben. Außerdem gestaltet sich die spätere Realisierung meiner Methode einfacher als die automatische explizite Erzeugung von Arbeitseinheiten und ihre Zuordnung zu Arbeitsprozessen. Der Erfolg in der Praxis gibt meinem Ansatz recht.

4.3.2 Strategiewahl anhand von Laufzeitgrößen

Eine gute Parallelisierungsstrategie erzeugt eine nicht zu große Anzahl von nicht zu kleinen Threads. Wenn man die genaue Größe (also Laufzeit) aller Teilprobleme statisch feststellen könnte, wäre die a priori Auswahl einer geeigneten Strategie möglich. Da dies bei irregulären Problemen aber nicht der Fall ist, müssen dynamisch gewonnene Kenngrößen des Problems verwendet werden. Ein Probelauf des Programms mit repräsentativen Daten liefert dabei die nötige Information.

Dabei sind Heisenberg-Effekte zu vermeiden, d.h. die Messung der Kenngrößen darf nicht so aufwendig sein, daß sie den Ablauf des Programms verfälscht. Außerdem muß eine Kenngröße für den Zweck der Strategiewahl leicht quantifizierbar und automatisch auswertbar sein — eine Aussage wie „der Baum ist regelmäßig aufgebaut“ ist zu unpräzise, und „das dritte Kind jedes Knotens hat doppelt so viele Unterknoten wie das zweite“ ist zwar präzise, aber kaum allgemein automatisch auswertbar.

4.3.2.1 Kenngrößen

Ein rekursives Programm bietet nach diesen Überlegungen folgende Kenngrößen an:

Rekursive Prozeduren: Welche Prozeduren rekursiv sind, läßt sich dank der Annahme der Funktionszeiger-Freiheit statisch aus dem Programmcode bestimmen.

Laufzeit: Die sequentielle Laufzeit t_{seq} des Problems ist Basis der Beschleunigungsmessungen und erlaubt Granularitätsaussagen.

Rekursionstiefe: Die erreichte maximale Tiefe $T(b)$ des Rekursionsbaums b gibt Aufschluß über die Größe des Problems. Die Verteilung der Rekursionstiefen der Unterbäume ist ein Maß für die Unregelmäßigkeit.

Verzweigungsgrad: Wie die Rekursionstiefe beschreibt der Verzweigungsgrad $V(b)$ des Rekursionsbaums b dessen Regelmäßigkeit. Außerdem erlaubt ein fester Verzweigungsgrad (V ist immer 0 oder X) weitere Optimierungen in der Threaderzeugung, wie später genauer beschrieben wird.

Knoten- und Blattzahl: In Verbindung mit der sequentiellen Laufzeit erlauben diese Zahlen ($|Knoten(b)| - |Blätter(b)|$ und $|Blätter(b)|$) eine Abschätzung der Granularität. Wenn die durchschnittliche Laufzeit pro Blatt z.B. wesentlich kleiner ist als der Aufwand zur Threaderzeugung, dann sind Strategien zu vermeiden, die kleine Restbäume und Blätter als eigene Threads starten würden.

Die Knotenzahl entspricht dem Zähler der Prozeduraufrufe bei konventionellen Laufzeitprofilen.

Rekursionsbaum: Wenn die Granularität des Problems es erlaubt, die Datengewinnung und Menge der Daten also nicht übermäßig groß sind, können alle Rekursionsbäume eines Programmlaufs aufgezeichnet werden. Der Rekursionsbaum ist die genaueste Basis für eine Strategiewahl, aber auch nicht perfekt, weil folgende Programmläufe mit anderen Daten zu anderen Bäumen führen können.

Eine Anreicherung des Baums mit Informationen über die Rechenzeit jedes Teilbaums würde hilfreich sein, scheitert bei feingranularen Problemen aber an der erzielbaren Genauigkeit der Zeitmessung.

Es ist möglich, Rekursionstiefe t, Verzweigungsgrad v und Knotenzahl z miteinander tabellarisch zu kombinieren, indem man für jede Prozedur P aufzeichnet, welcher Verzweigungsgrad in welcher Rekursionstiefe wie oft auftritt. Es ergibt sich also eine Tabelle $z_P(t, v)$ der entsprechenden Zähler — ein konkretes Beispiel findet sich später in Abbildung 5.4. Diese Tabelle bildet einen kompakten Kompromiß zwischen der informationsarmen globalen Rekursionstiefe und dem eventuell sehr großen exakten Rekursionsbaum. Aus einer Tabelle läßt sich der Rekursionsbaum mit gewissen Informationsverlusten gegenüber dem exakten Baum rekonstruieren, wie Abschnitt 5.6.2 im Realisierungskapitel darstellt.

Wenn mehrere Iterationen über eine rekursive Prozedur P auftreten, d.h. P mehrfach von einer nichtrekursiven Prozedur aufgerufen wird, schlägt sich das im tabellarischen Profil durch mehrere Einträge auf Rekursionstiefe Null nieder. Im Falle der kompletten Baumaufzeichnung wird für jeden externen Aufruf ein neuer Rekursionsbaum erzeugt.

Aus den aufgeführten Kenngrößen lassen sich, wie erwähnt, weitere Charakteristika ableiten, etwa die Granularität als $T_{seq}/Blätter(b)$. Dabei wird angenommen, daß sich die Blätter des Baums in ihrer Ausführungszeit nicht wesentlich unterscheiden — eine Detailbetrachtung einzelner Blatt-Zeiten ist in der Realität nicht nötig, wie die Ergebnisse in Kapitel 6 zeigen.

Andere Größen wie der Umfang der Eingabedaten sind hier nicht aufgeführt, da sie als „black box parameter“ für alle folgenden Programmläufe ähnlich sind und sich zudem automatisch kaum bestimmen lassen. Leichter meßbare verwandte Größen wie der Speicherbedarf des Problems tragen nicht sinnvoll zur Strategiewahl bei.

Außerdem gibt es noch Kenngrößen der Maschine wie Prozessorgeschwindigkeit, Zahl der Prozessoren oder Kosten für die Threaderzeugung, die eine genauere Definition von Begriffen wie „grobgranular“ überhaupt erst ermöglichen und daher in die Strategiewahl mit eingehen.

4.3.2.2 Strategiewahl

Basierend auf diesen Kenngrößen soll nun eine Parallelisierungsstrategie unter den möglichen 50 bis 100 Kombinationen ausgewählt werden. Die automatisch gewählte Strategie soll möglichst nahe an die Leistung der real optimalen Strategie herankommen. Für die Strategiewahl bieten sich mehrere Methoden an:

Heuristiken: Die Kenngrößen werden ggf. weiter aufbereitet, etwa durch Klassifikation des Rekursionsbaums oder Operationen auf dem Verzweigungsprofil. Ein Satz von Regeln bestimmt daraufhin, welche Parallelisierungsstrategie am erfolgversprechendsten ist.

Der Vorteil ist, daß eine Entscheidung aufgrund von Heuristiken schnell und ohne großen Aufwand erfolgen kann. Nötig für diesen Ansatz ist ein klarer und kleiner Satz von Regeln, um eine Feinabstimmung auf die verwendeten Benchmarks zu verhindern und die allgemeine Verwendbarkeit zu fördern.

Simulation: Auf einem aufgezeichneten Rekursionsbaum wird das Verhalten einer parallelen Ausführung mit verschiedenen Parallelisierungsstrategien simuliert. Die Strategie, die in der Simulation am besten abschneidet, wird für die Parallelisierung verwendet.

Die Simulation kann genauer auf das reale Verhalten des Programms eingehen als eine Heuristik. Für einen erfolgreichen Einsatz einer Simulation ist aber nötig, daß die Strategien, die in der Simulation am besten abschneiden, auch in der Realität eine sehr gute Beschleunigung erreichen. Zudem sollte die Laufzeit der Simulation deutlich geringer sein als die Laufzeit des eigentlichen Programms, da das System sonst statt einer Simulation einfach alle Parallelisierungsmöglichkeiten real durchprobieren könnte.

Maschinelles Lernen: Die Automatisierung der Datensammlung eröffnet die Möglichkeit, die für maschinelles Lernen nötigen Datenmengen bereitzustellen. Das System lernt dazu an einer Reihe von Benchmarks die für alle möglichen Strategien erzielten Beschleunigungen und kann danach Vorhersagen der Beschleunigung unbekannter Programme treffen.

Problem dieser Methode ist, das System mit einem breiten Bereich von Daten einzulernen und vor allem, unbekannte Programme automatisch so zu klassifizieren, daß ihr Verhalten aus einer Kombination der bekannten Benchmarks abgeleitet werden kann. Ein Auswendiglernen der Benchmarks ist zu vermeiden.

Leistungsmodellierung: Eine Modellierung der Leistung aller Strategien für ein gegebenes Programm, d.h. der erreichten Beschleunigungen, kann ebenfalls als Auswahlkriterium für eine Strategie dienen (eine Einführung in die Leistungsanalyse findet sich in [41]).

Notwendig dazu ist ein allgemeines Modell für die Beschleunigung rekursiver Programme, z.B. basierend auf dem Rekursionsbaum. Aus dem Baum könnten die von den Strategien erzeugten Arbeitseinheiten abgeleitet und etwa in einem Warteschlangenmodell, das die Threadabarbeitung beschreibt, analysiert werden. In sich geschlossene Formeln zur Leistungsvorhersage erscheinen angesichts des komplexen Problems kaum möglich. Eine alternative Modellierung wäre die Simulation des Threadablaufs, wie sie oben bereits beschrieben wurde, d.h. die Simulation kann konzeptionell der Leistungsmodellierung zugeordnet werden.

Die Herausforderung besteht im Aufstellen des allgemeinen Modells, das für alle rekursiven Programme gelten muß. Außerdem darf die Laufzeit der Analyse nicht zu hoch

im Vergleich zur Programmlaufzeit selbst werden, was bei aufwendigen Warteschlangenmodellen nicht immer garantiert werden kann.

Für das System wurde ein kombinierter Ansatz von Heuristiken und Simulation gewählt, nachdem sich anfängliche Ansätze mit maschinellem Lernen als weniger erfolgversprechend herausgestellt hatten (siehe Anhang A.2).

Die Möglichkeit, eine Strategie erst zur Laufzeit des Programms zu wählen und dem Programmverhalten dynamisch anzupassen, wurde nicht weiter verfolgt: Zum einen würde eine solche dynamische Anpassung wesentlich komplexeren Programmcode erfordern, der die Laufzeit des Programms stark verfälschen könnte — die in dieser Arbeit realisierten Strategien verwenden hingegen nur einen Vergleich von zwei ganzen Zahlen. Abhilfe schaffen könnte eine Strategiewahl, die nur alle n Programmschritte durchgeführt wird, was aber das schwierige Problem aufwirft, einen „Programmschritt“ automatisch zu erkennen und seine Granularität automatisch einzuschätzen. Zum anderen sind manche Strategien auf Daten angewiesen, die andere Strategien gar nicht zur Verfügung haben, etwa die Thread-Zähler bei den allgemeinen Strategien. Ein Wechsel von Tiefenstrategie zu allgemeiner Strategie wäre damit nicht möglich, oder aber die Tiefenstrategie müßte ebenfalls Zähler mitführen, was wie erwähnt zu Leistungseinbußen für kritische Abschnitte führen kann. Des weiteren sind viele der aufgezeichneten Daten erst nach dem Programmablauf aussagekräftig, z.B. läßt sich die maximale Rekursionstiefe erst genau bestimmen, wenn das Programm beendet ist.

Aus diesen Gründen untersucht die weitere Arbeit nur zur Laufzeit konstante Strategien, die sich jedoch in der Praxis gut bewähren und sehr gute Beschleunigungen erziehen, wie Kapitel 6 zeigt.

4.3.3 Instrumentierung zur Datengewinnung

Nachdem im vorigen Abschnitt aufgeführt wurde, welche Laufzeit-Informationen des Programms für die Strategiewahl nützlich sein können, bleibt die Frage, wie diese Informationen automatisch gewonnen werden können. Ein Automatismus zur Datensammlung (*profiling*) ist nötig, da das Gesamtsystem automatisch arbeiten soll und es dem Benutzer des Systems nicht zugemutet werden kann, sein Programm selbst um eine Datensammlungskomponente zu erweitern. Außerdem garantiert eine automatische Sammlung auch konsistente Datenformate und damit eine einfache Weiterverarbeitung.

4.3.3.1 Anforderungen und Entwurfsentscheidungen

Die Grundidee der automatischen Datengewinnung ist, den Quellcode des zu untersuchenden Programms so zu erweitern, daß die gewünschten Daten während des Programmlaufs gesammelt und ausgegeben werden — den Vorgang der Quellcodeänderung nennen wir *Instrumentierung*. Folgende Anforderungen stellen sich in diesem Kontext:

- Die Datensammlung muß automatisch in den Quellcode des Programms einzubringen sein (eine Instrumentierung des Binärcodes wird nicht weiter betrachet, da sie mit viel weniger Informationen auskommen müßte, als der Quellcode sie bietet, und die nötige Infrastruktur erheblich aufwendiger und maschinenabhängiger wäre).

- Sie darf die Semantik des Programms nicht verändern.

- Sie darf die Programmlaufzeit nicht stark verfälschen — eine Bestimmung der Laufzeit, die selbst die Laufzeit z.B. durch Zerstören des Cache-Inhalts wesentlich erhöht, ist offensichtlich nicht akzeptabel.

- Die gewonnenen Daten müssen automatisch weiterzuverarbeiten sein.

- Es sollte bei Bedarf möglich sein, verschiedene Grade an Datensammlung zu wählen, um z.B. interessante Prozeduren des Programms detaillierter zu analysieren oder uninteressante Teile auszublenden.

Vor diesem Hintergrund stellen sich die Entwurfsentscheidungen:

Durchführende Systemkomponente: Die Instrumentierung kann durch einen speziell erweiterten Übersetzer (*compiler*) oder durch ein vom Übersetzer unabhängiges Programm erfolgen.

Vorteil des Übersetzers ist, daß dieser über weitgehende semantische Information verfügt, z.B. alle Prozeduraufrufe und ihre Parameter kennt. Der Nachteil ist pragmatischer Natur — frei erhältlicher Quellcode zu Übersetzern, etwa `gcc`, ist historisch gewachsen und erfordert lange Einarbeitung, bevor er produktiv verwendbar ist. Außerdem legt man sich mit der Wahl eines Übersetzers in der Infrastruktur sehr fest.

Ein eigenständiges Programm, das den Quellcode des zu untersuchenden Programms einliest, ihn manipuliert und den resultierenden Quellcode ausgibt, ist hingegen unabhängig vom verwendeten Übersetzer und damit portabler. Als Systemkomponente ist es außerdem übersichtlicher und voraussichtlich leichter zu entwickeln. Nachteil ist, daß eventuell schon im Übersetzer vorhandene Informationen nochmals ermittelt werden müssen. Eine Unter-Entscheidung ist hierbei, ob ein kompletter Zerteiler (*parser*) mit zusätzlichen semantischen Analysen nötig ist, oder ob ein auf Syntax und Mustern basierter Ansatz ausreicht.

Infrastruktur: Es bietet sich an, eine Bibliothek mit Funktionen bereitzustellen, die die Infrastruktur der Datensammlung bilden. Diese Funktionen bieten z.B. spezielle Datenstrukturen an, fügen dem Laufzeitprofil Informationen hinzu oder geben es aus. Die Instrumentierung des Programms fügt dann an geeigneten Stellen solche Funktionsaufrufe ein.

Ausgabe: Die Ausgabe des Profils sollte nach Ende des eigentlichen Programms erfolgen, um den Ablauf nicht zu verfälschen. Sie kann entweder binär codiert in eine spezielle Datei geschrieben werden, die später von Auswertewerkzeugen interpretiert wird, oder textuell auf der Standardausgabe erfolgen. Letzterer Ansatz hat den Vorteil, daß während der Entwicklung des Systems die Arbeitsweise der Instrumentierung leichter überprüft werden kann und keine zusätzlichen Dateien nötig sind. Im letzteren Fall muß die durch das System erzeugte zusätzliche Ausgabe einfach automatisch auszufiltern sein, um eine eventuelle Weiterverarbeitung der Programmausgabe durch Fremdwerkzeuge nicht zu behindern.

Die Techniken der Instrumentierung, z.B. das Einfügen von Zählern für Prozeduraufrufe oder die Erweiterung von Prozeduren um einen Rekursionstiefen-Parameter, werden in Kapitel 5 genauer beschrieben.

4.3.3.2 Optionen

Die Instrumentierung eines Programms sollte automatisch erfolgen, aber an einigen Stellen kann es sinnvoll sein, Benutzereingriffe zu ermöglichen. Zum einen sind Annotationen des Programmcodes eine Möglichkeit, nicht integrierte Komponenten wie die Datenabhängigkeitsanalyse zu ersetzen. Zum anderen möchte der Benutzer eventuell das Systemverhalten steuern. Auf der Entwurfsebene ergeben sich diese Parameter:

Prozedurauswahl: Das Standardverhalten des Systems ist, jede rekursive Prozedur zu instrumentieren, damit ohne Benutzereingriff alle Daten erfaßt werden. Für den Fall, daß einige Prozeduren uninteressant sind, soll sich die Datensammlung für sie gezielt abschalten lassen.

Baumaufzeichnung: Für eine genauere Analyse des Programms ist es sinnvoll, den gesamten Rekursionsbaum aufzuzeichnen. Dies sollte aber nur optional geschehen, um den zu erwartenden höheren Aufwand in Datensammlung und Datenmenge vermeiden zu können.

Zeitmessungen: Die Knotenlaufzeiten grobgranularer Rekursionen lassen sich auch mit der Zeitauflösung des Betriebssystems sinnvoll messen. Da die Granularität einer Prozedur nicht automatisch statisch zu ermitteln ist, sollte diese Zeitaufzeichnung optional erfolgen.

Außerdem kommen typische Optionen wie die Wahl eines mehr oder weniger detaillierten Fortschrittreports der Instrumentierung hinzu.

4.3.4 Parallelisierung für Threads

Die eigentliche Parallelisierung des Programms ähnelt technisch stark der Instrumentierung zur Datengewinnung. Auch hier wird Programmcode eingefügt, der die gewünschte Funktionalität erbringt: Abprüfen von Laufzeitbedingungen der Strategiewahl, Erzeugen von rekursiven Aufrufen als Threads, Einsammeln der Threads nach Ende der Berechnung usw.

4.3.4.1 Anforderungen und Entwurfsentscheidungen

Die Anforderungen bezüglich Automatisierung und Semantikerhalt decken sich mit denen an die Instrumentierung. Auch bei der Systemkomponente stellen sich die gleichen Alternativen. Als Infrastruktur werden die Thread-Bibliotheken des Betriebssystems verwendet.

4.3.4.2 Optionen

Wie schon bei der Instrumentierung soll es auch bei der Parallelisierung Möglichkeiten geben, das Systemverhalten zu beeinflussen. Dabei ist darauf zu achten, daß die Standardeinstellung ohne Benutzereinwirkung bereits in einer guten Parallelisierung resultiert:

Prozedurauswahl: Normalerweise werden alle rekursiven Prozeduren parallelisiert. Es ist aber sinnvoll, einzelne Prozeduren davon auszuschließen, wenn sie z.B. sequentialisierende Datenabhängigkeiten enthalten, nicht den Kernteil der Berechnung darstellen und daher uninteressant sind, oder sie zu feinkörnig für eine effiziente Parallelisierung sind.

Warten auf Teilberechnungen: Da das System keine Datenabhängigkeitsanalyse enthält, sondern auf existierende Analysen aufbaut, muß eine Schnittstelle zu diesen Analysen gegeben sein. Sie kann etwa durch Programmannotationen realisiert werden. Ohne solche Annotationen fügt das System direkt vor dem Ende einer Prozedur Code ein, der die Ergebnisse von in der Prozedur gestarteten Threads aufsammelt. Wenn diese Ergebnisse bereits früher benötigt werden, soll eine entsprechende Annotation das dem System mitteilen können.

Wahl der Prozessorzahl: Beim Start des parallelen Programms sollte angegeben werden können, wieviele der vorhandenen CPUs eines Mehrprozessorrechners verwendet werden sollen. Damit kann anderen Benutzern Rechenleistung reserviert werden. Die Grundeinstellung ist, alle verfügbaren CPUs zu belegen.

Außerdem gibt es Optionen zur weiteren Optimierung, wie das Verhindern einer Threaderzeugung für den letzten rekursiven Aufruf in einer Prozedur, die im Implementierungskapitel genauer beschrieben werden.

4.4 REAPAR Systemüberblick

In Hinblick auf diese Entwurfsentscheidungen ergibt sich ein Strukturbild des Gesamtsystems, das in Abbildung 4.2 gezeigt ist.

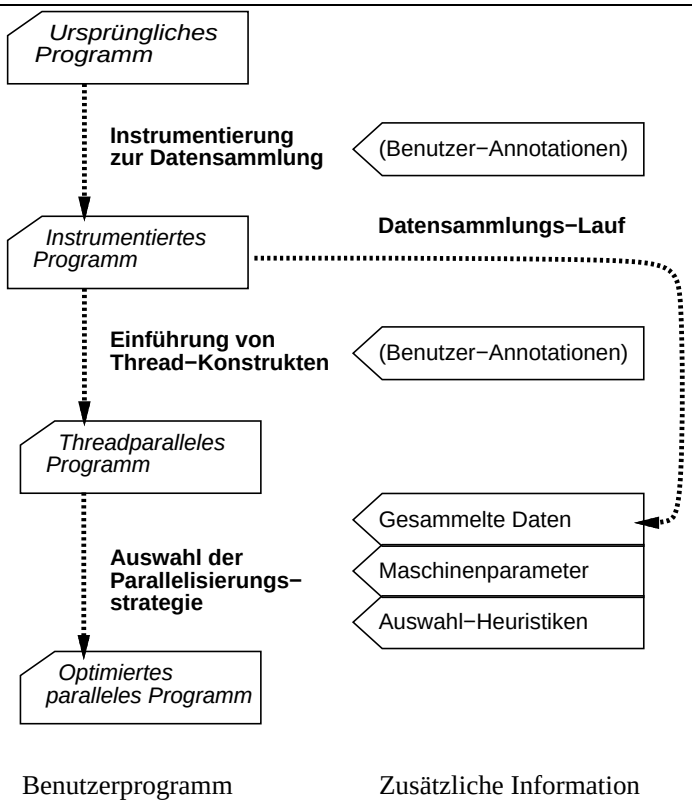


Abbildung 4.2: Ablauf des REAPAR Systems

Der Quellcode des Programms wird durch das System automatisch instrumentiert und parallelisiert, ggf. unter Berücksichtigung von Annotationen des Benutzers. Beim Meßlauf

sammelt das Program die nötigen Daten, um eine Auswahl der Parallelisierungsstrategie vorzunehmen. Das optimierte parallele Programm wird dann auf ähnlichen Datensätzen mehrfach aufgerufen.

Die Datensammlung kann prinzipiell auch am bereits parallelisierten Programm erfolgen — für diesen Zweck würde eine möglichst allgemeine Strategie gewählt, die voraussichtlich eine gewisse, wenn auch nicht optimale Beschleunigung erzielt. Allerdings ist dann z.B. eine detaillierte Aufzeichnung des Rekursionsbaums technisch schwierig, da ein paralleler Aufbau der entsprechenden Datenstruktur zu Schreibkonflikten führt und das Einführen kritischer Abschnitte die parallele Beschleunigung bei allen außer den grobgranularsten Programmen zunichte macht.

Die Benutzer-Annotationen stellen zugleich eine Schnittstelle zu anderen Systemen dar. Eine Komponente, die eine Datenabhängigkeitsanalyse des Quellcodes durchführt und dabei ermittelt, an welchen Stellen des Programms die Ergebnisse vorheriger rekursiver Aufrufe benötigt werden, kann dieses Wissen als Annotation in den Quellcode einbauen. Für das REAPAR System ist es unerheblich, ob die Annotation manuell oder durch weitere Systemkomponenten erstellt wurde; dadurch ist das System leicht erweiterbar.

Kapitel 5

Realisierung

Nachdem die Anforderungen und der grundlegende Entwurf des Systems bekannt sind, beschreibt dieses Kapitel die konkrete Realisierung und Implementierung. Dazu gehören die gewählten Methoden zur Verwirklichung des Entwurfs, Hilfsmittel, die den Umfang dieser Arbeit in realistischen Grenzen halten, die Erklärung der realisierten Systemkomponenten und eine Beschreibung der Einschränkungen des Systems.

Dieses Kapitel umfaßt mit der Beschreibung der Instrumentierung, der Parallelisierung und der Strategiewahl den technischen Kernteil der Arbeit.

5.1 Szenario der Anwendung

Bevor wir auf die Details der Realisierung eingehen, soll das folgende Anwendungsszenario einen Eindruck des Systems und des typischen Einsatzes geben:

Ein Anwender ohne Erfahrung mit der Parallelisierung von Programmen hat Zugriff auf einen Mehrprozessorrechner mit gemeinsamem Speicher. Er[1] weiß aber, daß das REAPAR-System Rekursionen parallelisiert, um eine beschleunigte Ausführung auf Parallelrechnern zu erzielen. Daher schreibt er seinen Algorithmus zur Galaxiensimulation in C und formuliert ihn in der natürlichen rekursiven Weise, anstatt ihn iterativ umzuschreiben. Er ist sich sicher, daß die einzelnen rekursiven Aufrufe der Simulation auf unabhängigen Daten arbeiten. Rekursive Prozeduren zum Aufbau der Galaxiendaten, die nicht parallelisiert werden sollen, weil sie erwartungsgemäß einen vernachlässigbaren Teil der Gesamtlaufzeit ausmachen und schreibend auf gemeinsame Daten zugreifen, annotiert er im Quellcode als nicht parallel.

Nach abgeschlossener Fehlersuche auf einem Einprozessorrechner übergibt der Benutzer seinen Quellcode und die Eingabeparameter für den Testlauf des Programms an das REAPAR-System. Das System instrumentiert automatisch den Code, übersetzt ihn und führt ihn sequentiell aus. Dabei wird das tabellarische Laufzeitprofil aufgezeichnet und nach Programmende automatisch analysiert. Die Analyse ergibt, daß auf ein Blatt bezogene Laufzeit sehr klein ist. Daher verwendet REAPAR eine Heuristik, die einen guten Parameter für die Tiefenstrategie zur Parallelisierung des Programms auswählt — für eine allgemeine Strategie ist das Problem zu feingranular. Außerdem bemerkt das System, daß sich das Programm entweder achtmal oder nie rekursiv verzweigt. Es empfiehlt dem Benutzer daher, eine Nothread-Annotation einzufügen (s.u.), die den parallelen Ablauf weiter beschleunigen kann.

[1] oder *sie* — durch konsistente Verwendung des männlichen Pronomens in dieser Arbeit sollen keinesfalls antiemanzipatorische Denkweisen gefördert werden!

Schließlich parallelisiert das System das Programm durch automatisches Einfügen von Threaderzeugungen im Quellcode und übersetzt es. Der Benutzer kann nun das resultierende ausführbare parallele Programm für seine nächsten 30 Eingabedaten verwenden, die wie die Testdaten Kugelsternhaufen darstellen, d.h. eine ähnliche Grundstruktur aufweisen.

Durch die Verwendung von REAPAR nutzt er also die Vorteile eines beschleunigten parallelen Ablaufs aus, ohne über Kenntnisse der Parallelisierung zu verfügen. Das notwendige Wissen um Datenabhängigkeiten, die laut REAPAR-Handbuch einem parallelen Ablauf im Wege stehen, besitzt er als Autor des Programms bereits, und das System gibt ihm zusätzlich automatisch Hinweise, wie er die Leistung noch weiter steigern kann.

Für viele Programme ohne Datenabhängigkeiten sind noch nicht einmal Annotationen nötig, weil das Standardverhalten des Systems bereits eine gute Parallelisierung bewirkt.

5.2 Techniken und Werkzeuge

Das gesamte REAPAR System wurde unter UNIX (Solaris 2.6) entwickelt. Zum Einsatz kamen auf der C-Seite die Übersetzer von Sun und der frei verfügbare `gcc` sowie der Quellcode-Debugger `gdb`. Die Analyse- und Quellcodetransformationswerkzeuge wurden, wie unten erläutert, in `perl` geschrieben. Für übergreifende Kontrollstrukturen, z.B. die Steuerung einer Meßreihe über alle Parallelisierungsstrategien hinweg, und zur Auswertung der Ergebnisse fanden verschiedene Scripte in `sh`, `csh` und `awk` Verwendung.

Die Parallelisierung findet wie erwähnt durch die Verwendung von Threads statt rekursiver Aufrufe statt. Es gibt die POSIX-Norm „IEEE Standard 1003.1c-1995“ für die Programmierschnittstelle von Threads, so daß die Portabilität von Thread-Programmen gewährleistet ist. Die in dieser Arbeit verwendeten Solaris-Threads [90] sind funktional äquivalent zu den POSIX-Threads [89], bieten zusätzlich aber noch eine weitere Hierarchieebene: der Programmierer kann bestimmen, durch wieviele grundlegende Betriebssystemobjekte (LWPs) die Threads ausgeführt werden, und damit den potentiellen Parallelismus des Programms steuern, z.B. nur sechs von acht verfügbaren Prozessoren einer Maschine belegen.

5.3 Annotationen

Im vorangehenden Kapitel und im obigen Szenario wurde die Möglichkeit aufgezeigt, Annotationen des Quellcodes als Eingriffsmöglichkeit des Benutzers und als Schnittstelle zu anderen Werkzeugen zu verwenden. Das REAPAR System unterstützt die folgenden Annotationen in Form von C-Kommentaren:

`/* NOPARALLEL */`

Die Prozedur, in oder vor der diese Annotation steht, wird nicht parallelisiert. Einsatzbereiche sind Prozeduren, deren rekursive Äste nicht unabhängig voneinander sind, oder Prozeduren in unteren Ebenen einer Rekursionshierarchie, die zu feinkörnig für eine lohnende Parallelisierung sind.

Ohne diese Annotation parallelisiert das System alle rekursiven Prozeduren (mit den in Abschnitt 5.7 aufgeführten Einschränkungen).

Das REAPAR Werkzeug `hierarchies_check_simulation` gibt automatisch anhand eines Rekursionsbaums Empfehlungen, bei welchen Prozeduren in tieferen Hierarchieebenen sich eine solche Annotation lohnen könnte, siehe Abschnitt 5.6.3.6.

`/* NEEDRESULTS */`

Normalerweise werden alle Threads, die in einer Prozedur anstelle von rekursiven Aufrufen erzeugt wurden, kurz vor Verlassen der Prozedur wieder eingesammelt (d.h. vor jedem `return` und am Prozedurende). Wenn von diesen Threads berechnete Daten schon früher in der Prozedur benötigt werden, kann durch Verwendung dieser Annotation eine zusätzliche Stelle zur Threadzusammenführung eingefügt werden. In den Programmzeilen nach der Annotation sind dann alle vor ihr in der Prozedur erzeugten Threads beendet.

Diese Annotation kann z.B. von einem Datenabhängigkeits-Analysewerkzeug eingefügt werden.

`/* NOTHREAD */`

Diese Annotation veranlaßt, daß der ihr folgende rekursive Aufruf nicht als Thread parallelisiert wird. Dies kann die Leistung des Programms erhöhen, wenn eine statische Zahl von rekursiven Aufrufen in einer Prozedur vorliegt und den Aufrufen großen Berechnungen mehr folgen. In diesem Fall lohnt es sich, den letzten rekursiven Aufruf nicht als Thread, sondern sequentiell durchzuführen, was eine Threaderzeugung einspart.

Abschnitt 6.2.7 zeigt, daß manche Benchmarks durch die Nothread-Annotation einen deutlichen Leistungsgewinn verbuchen können.

Standardverhalten des Systems ist die Parallelisierung aller rekursiven Aufrufe in einer Prozedur. Das System erkennt durch Analyse der Laufzeitdaten, ob eine Prozedur von dieser Annotation profitieren könnte, und gibt dem Benutzer einen entsprechenden Hinweis — ein automatisches Einfügen der Annotation wäre angenehmer zu benutzen, aber ist mit den technischen Gegebenheiten nur schwer zu realisieren.

`/* NOSTATS */`

Ist genauere Information über eine rekursive Prozedur für den Benutzer uninteressant, so kann er mit dieser Annotation vor oder in der Prozedur bewirken, daß für sie keinerlei Statistiken gesammelt werden. Stattdessen werden lediglich ihr Rekursionstiefe-Parameter hinzugefügt, aber keine Profile gesammelt oder (im Falle der Baumaufzeichnung) Rekursionsbäume aufgezeichnet.

Diese Annotation kann den Speicherbedarf von feingranularen Programmen bei der Baumaufzeichnung drastisch reduzieren, z.B. wenn der Baum rekursiv aufgebaut wird aber die entsprechende Prozedur gar nicht für die Parallelisierung von Interesse ist. Auch die Aufzeichnung feingranularer Prozeduren in unteren Ebenen des Rekursionsbaums läßt sich hiermit verhindern.

Eine Nostats-annotierte Prozedur darf bei aktivierter Baumaufzeichnung ihrerseits keine rekursive Prozedur aufrufen, für die Statistiken gesammelt werden, da die entsprechenden Baumparameter durch die Annotation nicht mehr zur Verfügung stehen. In diesem Fall bricht die Instrumentierung mit einer entsprechenden Warnung ab.

Standardverhalten des Systems ist die Profil- bzw. Baumaufzeichnung für alle rekursiven Prozeduren, es sei denn, die unten beschriebene `-autonostats` Option wird verwendet und aktiviert die Datensammlung nur für parallelisierte Prozeduren.

Needresults und Noparallel sind Annotationen, die für den korrekten parallelen Ablauf eines Programms notwendig sein können. Bei einem Programm, das keine Datenabhängigkeiten

zwischen rekursiven Prozeduren aufweist und die Ergebnisse einer rekursiven Berechnung nicht in der selben Prozedur weiterverwendet, sind Annotationen überflüssig. Von den fünf Benchmarks, die bei der Systementwicklung einflossen, benötigen nur zwei (Barnes Hut und Power) die Noparallel-Annotation für rekursive Baumaufbau-Prozeduren, die sich nicht ohne weiteres parallel ausführen lassen. Zwei Benchmarks (Power und Queens) verwenden die Needresults-Annotation, um Ergebnisse von rekursiven Aufrufen rechtzeitig bereitzustellen.

Im Gegensatz zu den beiden ersten Annotationen ändern die Nothread- und Nostats-Annotationen die Semantik des Programms nicht, sondern führen nur zu einer Leistungssteigerung. Zwei Benchmarks (Eigenvalue und Fractal) haben einen festen Verzweigungsgrad ihrer rekursiven Prozedur und profitieren von der Nothread-Annotation. Beim Barnes Hut Benchmark mit feingranularer unterer Rekursionshierarchie und nichtparallelen rekursiven Hilfsprozeduren verringert die Nostats-Annotation das Datenaufkommen der Baumaufzeichnung signifikant.

Kapitel 6 beschreibt die Benchmarks und die Auswirkung der Annotationen im Detail.

Zusätzlich zu den Annotationen bieten Instrumentierung und Parallelisierung auch die Option, automatisch die Statistiksammlung für alle nicht parallelisierten Prozeduren auszuschalten, um die Datengewinnung auf die parallelen Prozeduren zu fokussieren.

5.4 Instrumentierungskomponente

Nach der allgemeinen Betrachtung der Annotationen beschreiben die folgenden Abschnitte die Systemkomponenten von REAPAR. Dabei liegt der Schwerpunkt auf den Funktionsprinzipien und der Umsetzung der Entwurfsentscheidungen — für eine genaue Beschreibung der Benutzerschnittstelle und der Programmdetails sei auf den entsprechenden technischen Bericht [43] verwiesen.

5.4.1 Instrumentierung

Die Systemanforderungen besagen, daß Rekursionstiefe, Verzweigungsgrad und Knoten/Blattzahl der rekursiven Prozeduren eines Programms automatisch ermittelt werden müssen. Dazu wird in jedem rekursiven Aufruf gemessen, in welcher Rekursionstiefe der Aufruf erfolgt und wie oft sich die entsprechende Prozedur rekursiv verzweigt. Die Ergebnisse werden in einer Tabelle von Rekursionstiefe×Verzweigungsgrad festgehalten und am Programmende ausgegeben.

Die Gesamtlaufzeit des Programms wird durch Ablesen der Uhrzeit mit `gettimeofday()` zu Beginn und Ende des Programms festgestellt, die verbrauchte Benutzer- und Systemzeit durch den Systemaufruf `getrusage()`. Abgeleitete Kenngrößen des Programms wie die Granularität werden erst in den weiterverarbeitenden Werkzeugen analysiert und müssen nicht mit aufgezeichnet werden.

5.4.2 Prinzip

Der Quellcode des Programms wird für jede rekursive Prozedur um zwei Zähler erweitert. Ein Zähler t ist ein Prozedurparameter und mißt die Rekursionstiefe. Er wird beim initialen Aufruf der Prozedur mit Null belegt und bei jedem rekursiven Aufruf um eins erhöht. Der andere Zähler v wird zu Beginn der Prozedur auf Null gesetzt und bei jeder rekursiven

Verzweigung um eins erhöht. Er mißt den aktuellen Verzweigungsgrad, wobei ein Grad von Null einem Blatt im Rekursionsbaum entspricht.

Vor Verlassen der Prozedur, d.h. vor jedem `return` und direkt vor dem textuellen Ende der Prozedur, werden die beiden Zähler gespeichert. Dazu wird in der zur Prozedur gehörigen Tabelle der Eintrag an der Stelle (t,v) erhöht, um anzuzeigen, daß die Prozedur sich v mal in der Tiefe t verzweigt hat. Die Einträge sind kumulativ.

Zur Unterstützung der Statistikerfassung wurde eine Programmbibliothek erstellt, deren Funktionen besagte Tabellen initialisieren, Zählerstände sichern und die ermittelten Werte am Programmende ausgeben. Die Instrumentierung fügt lediglich Aufrufe an diese Funktionen ein.

Abbildung 5.1 zeigt einen Codeauszug einer rekursiven Funktion vor und nach der Instrumentierung.

5.4.3 Algorithmus

Der Algorithmus der Programminstrumentierung ist in Abbildung 5.2 angegeben. Rekursive Prozeduren werden zuvor identifiziert, indem das System in zwei Quellcode-Durchläufen zunächst alle Prozeduren des Programms bestimmt und dann alle Aufrufe an diese Prozeduren ermittelt. Eine Prozedur ist rekursiv, wenn sie selbst in der transitiven Hülle ihrer Aufrufrelation vorkommt.

Die Komplexität der Prozedurbestimmung ist linear in der Zahl der Programmzeilen, ebenso die Aufrufbestimmung (das Feld mit den Prozedurinformationen ist als Hashtabelle realisiert). Die eigentliche Instrumentierung hat eine Komplexität von $O(R * C)$, wobei R die Zahl aller rekursiven Prozeduren und C die Zahl der Aufrufe an eine rekursive Prozedur ist. Zeitmessungen hierzu finden sich im Abschnitt 6.2.9.

Vereinfachend wird hier angenommen, daß der gesamte relevanter Quellcode in einer einzigen Datei vorliegt.

5.4.4 Baumaufzeichnung

Zusätzlich zur normalen Datensammlung kann auch Code ins Programm eingefügt werden, der den kompletten Rekursionsbaum aufzeichnet, optional mit den Laufzeiten aller Knoten. Das Vorgehen dazu ist völlig analog: Es wird eine globale Datenstruktur angelegt, die bei jedem rekursiven Aufruf um einen entsprechenden Knoten erweitert wird. Die Prozedurparameter werden um den aktuellen Knoten im Baum und die Baumdatenstruktur selbst ergänzt. Bei einer Zeitmessung wird zusätzlich am Beginn und Ende einer Prozedur die Systemzeit abgelesen und im Knoten der Prozedur vermerkt, was aus Betriebssystemgründen nur mit einer Auflösung von 50Hz möglich ist.

Auch Iterationen über rekursive Prozeduren, wie sie in mehrphasigen Programmen vorkommen, werden erfaßt. Bei jedem initialen rekursiven Aufruf wird dazu ein neuer Rekursionsbaum angelegt. Alle aufgezeichneten Rekursionsbäume aller Prozeduren werden am Programmende ausgegeben.

Wie in Kapitel 4.3.3 gefordert, wird die von REAPAR erzeugte Ausgabe in maschinell leicht ausfilterbare Blöcke eingebettet, was die Extraktion der Daten durch REAPAR vereinfacht und zusätzlich eventuellen Fremdwerkzeugen, die die Programmausgabe weiterverarbeiten, eine Filterung ermöglicht.

```
...

void foo(double a, char *b)
{
  ...
  if (condition) {
    ...
    return;
  }
  ...
  foo(c, d);
  ...
  if (condition) {
    ...
    foo(e, f);
  }
  ...
}

...

main() {
  ...
  foo(a, b);
  ...
}
```

⇒

```
...
#include "profile.h"
...

void foo(int profile_depth_foo,
         double a, char *b)
{
  int branch_count=0;
  ...
  if (condition) {
    ...
    ProfileAddStat(profile_stat, 100,
                   profile_depth_foo,
                   branch_count, 1);
    return;
  }
  ...
  branch_count++;
  foo(profile_depth_foo + 1, c, d);
  ...
  if (condition) {
    ...
    branch_count++;
    foo(profile_depth_foo + 1, e, f);
  }
  ...
  ProfileAddStat(profile_stat, 100,
                 profile_depth_foo,
                 branch_count, 1);
}

...

main() {
  profile_stat = ProfileNewStats(300,40,5);
  ProfileInitLine(profile_stat,100,"foo");
  ...
  foo(0, a, b);
  ...
  ProfilePrint(profile_stat);
}
```

Abbildung 5.1: Beispielcode vor und nach der Instrumentierung, hinzugefügter Code kursiv hervorgehoben und leicht vereinfacht. Die Variable $profile\_depth\_foo$ entspricht dem Tiefenzähler t, $branch\_count$ dem Verzweigungszähler v im Text. 100 ist die Zeilennummer des Prozedurkopfs von `foo()`, 40 die maximal erwartete Rekursionstiefe, 5 der maximale Verzweigungsgrad und 300 die Zeilenzahl des Programms.

```
Information gathered during program analysis:
wherecalled[R] lists all calls to R with the position and procedure in which they occur
allcalls[R] lists all procedures called by R

Insert #include "profile.h" at the first line
Insert profile initialization code for the global profile and for each recursive procedure
at the beginning of main()
Insert atexit() handler for the final profile output
for all recursive procedures R
      Prepend recursion depth parameter profile_depth_R to the procedure's
      parameter list (in its header as well as in any declarations)
      Insert recursion branch counter branch_count to the procedure's variables
      for all return statements within R
            Insert profile information update before return:
                  ProfileAddStat(profile_stat, start line of R, profile_depth_R,
                  branch_count)
      end for
      Insert profile information update just before end of procedure
      for all calls C to R in wherecalled[R]
            if (C occurs within R)
                  prepend profile_depth_R + 1 to the call's parameters
            else if (C occurs within recursive procedure P in allcalls[R])
                  prepend profile_depth_P + 1 to the call's parameters
            else
                  prepend 0 to the call's parameters, starting the recursion depth counter
            end if
            if (C occurs within recursive procedure P in allcalls[R], including R)
                  Insert branch_count++ before call
            end if
      end for
end for
perform insertion changes
```

Abbildung 5.2: Algorithmus: Instrumentierung des Quellcodes für die Datensammlung zur Laufzeit. Eingabe ist der Quellcode und die Listen `wherecalled[]` und `allcalls[]`, Ausgabe ist der instrumentierte Quellcode.

5.4.5 Beispielsausgabe

Abbildung 5.3 gibt in gekürzter Form die Informationen wieder, die REAPAR bei der Instrumentierung des Eigenvalue-Benchmarks liefert. Deutlich sind die Phasen des Algorithmus sichtbar: Die Prozedurerkennung findet Prozedurnamen, Typen und Parameter, die dann zur Aufruf-Relation zusammengefügt werden. Anhand dieser Relation stellt die Rekursionserkennung dann die rekursiven Prozeduren fest. Diese Ausgaben von REAPAR sind auch ohne Verwendung des Restsystems nützlich, um einen Einblick in die Zusammenhänge eines unbekannten Quellcodes zu erhalten oder um „tote Prozeduren“ zu finden, die nirgends aufgerufen werden.

Um einen Eindruck der zur Programmlaufzeit aufgezeichneten Daten zu geben, ist in Abbildung 5.4 die Ausgabe der Datensammlung des Eigenvalue-Benchmarks für die Eingabegröße 8 abgedruckt. Wie man sieht, wird auch Zusatzinformation über die instrumentierten Prozeduren und ihre Aufrufe ausgegeben, die von der Strategiewahlkomponente weiterverwendet wird. Die tabellarische Profilausgabe erfolgt aufgrund ihrer Kompaktheit immer,

```
Found the following procedures:

  1) procedure rand                 from line   28 to   28
        type: extern int
        parameters:
  2) procedure srand                from line   29 to   29
        type: extern void
        parameters:
  3) procedure FillMatrixGeom       from line   62 to   90
        type: void
        parameters:
           int n
           double *d
           double *e
[...]
  9) procedure LoopBody_h           from line  205 to  239
        type: void
        parameters:
           Interval_t * new
 10) procedure Mimd                 from line  248 to  260
        type: void
        parameters:
           void
 11) procedure main                 from line  268 to  303
        type: void
        parameters:
           int argc
           char ** argv

Found the following calls:

  1) rand                 called in FillMatrixRnd|113,21:FillMatrixRnd|115,21:
  2) srand                called in main|275,9:
  3) FillMatrixGeom       called in main|294,28:
  4) FillMatrixUnif       called in main|295,28:
  5) FillMatrixRnd        called in main|293,27:
  6) Gerschgorin          called in Mimd|254,15:
  7) IeeeCount            called in LoopBody_h|214,15:
  8) IsLeaf               called in LoopBody_h|221,22:LoopBody_h|224,15:
  9) LoopBody_h           called in LoopBody_h|231,11:LoopBody_h|236,11:Mimd|258,14:
 10) Mimd                 called in main|301,8:
 11) main                 called in

Found the following recursive procedures:

  1) LoopBody_h

Instrumenting program...

Writing output to file 'eigenvalue_instrumented.c' ...
```

Abbildung 5.3: Beispielhafte Ausgabe während der Programmanalyse durch REAPAR: Prozeduridentifikation, Erstellung der Aufrufrelation und Identifikation rekursiver Prozeduren. Das Format der Aufrufrelation ist $f|z,s$: für einen Aufruf durch Funktion f in Quellcode-Zeile z Spalte s.

die genaue Baumaufzeichnung wird hingegen nur für grobgranulare Benchmarks aktiviert. Dementsprechend gibt es auch verschiedene Methoden zur Strategiewahl basierend auf Profilen (feingranular) bzw. Bäumen (grobgranular), wie später in Abschnitt 5.6 genau dargestellt wird.

```
BEGIN REAPAR TREES
  Procedure number 1 = LoopBody_h

  Recursion Tree for 'LoopBody_h', Number 1, Iteration 0:

  (1(1(1(1)(1))(1(1)(1)))(1(1(1)(1))(1(1)(1))))
END REAPAR TREES
BEGIN REAPAR RSCINFO
  Wallclock time = 0.09 seconds
  User time      = 0.03 seconds
  System time    = 0.09 seconds
END REAPAR RSCINFO
BEGIN REAPAR PROCINFO
  Procedure ( 1) LoopBody_h at line  205 calls :LoopBody_h:
END REAPAR PROCINFO
BEGIN REAPAR PROFILES

Profile:
  [...]
  205 - LoopBody_h
        (  0:    0    0    1    0  sum =    1)
        (  1:    0    0    2    0  sum =    2)
        (  2:    0    0    4    0  sum =    4)
        (  3:    8    0    0    0  sum =    8)
        (  4:    0    0    0    0  sum =    0)
        (sum:    8    0    7    0  SUM =   15)
  [...]
END REAPAR PROFILES
```

Abbildung 5.4: Beispielhafte Ausgabe eines Programmprofils (Erklärung im Text).

Diese Ausgabe liest sich wie folgt: Es gibt eine rekursive Prozedur namens `LoopBody_h`, laufende Nummer 1, die nur eine Iteration 0 hat, also einmalig von außerhalb aufgerufen wird. Ihr Rekursionsbaum ist in Infix-Notation aufgeführt — jede sich öffnende Klammer bedeutet einen neuen Knoten, die Zahl gibt die Prozedur des Knotens an, und eine schließende Klammer verläßt den Knoten. Wie für das kleine Beispielproblem nicht verwunderlich, besteht die kurze Laufzeit fast nur aus Systemzeit für die Ausgabe. Es folgt die Prozedurinformation mit der Aufrufsrelation, die eine automatische Weiterverarbeitung erlaubt.

Das Laufzeitprofil zeigt in seinen Spalten die Verzweigungsgrade beginnend mit Null, d.h. der linke Eintrag einer Zeile gibt die Zahl der Blätter in der entsprechenden Rekursionstiefe an, der nächste die der Blätter mit Verzweigungsgrad 1, dann die mit Grad 2 etc. Eine Zeile steht jeweils für eine Rekursionstiefe, beginnend mit Null an der Wurzel des Rekursionsbaums. Die Rekursionstiefe steht abgetrennt am Anfang der Zeile vor dem Doppelpunkt. Es gibt z.B. genau 8 Blätter (1. Spalte) auf Tiefe 3 und 4 Knoten mit Verzweigungsgrad 2 (3. Spalte) in Tiefe 2. Am Ende einer Zeile werden die in ihr enthaltenen Knoten und Blätter summiert. Abschließend im Profil wird über die Verzweigungsgrade und

Gesamtknotenzahlen summiert.

Man sieht, daß sich das Programm genau zweimal oder gar nicht verzweigt. Die Einträge stellen einen perfekten Binärbaum da, der in Ebene n genau 2^n Knoten hat und in der letzten Ebene nur aus Blättern besteht.

Mit dem REAPAR-Werkzeug `tree_graph_output` kann der aufgezeichnete Rekursionsbaum auch als Grafik in mehreren Formaten ausgegeben werden. Die Knoten verschiedener Prozeduren werden dabei in verschiedenen Grautönen eingefärbt. Optional kann für große Rekursionsbäume die Darstellung der Knoten abgeschaltet werden. Dieses Werkzeug ermöglicht einen genauen und übersichtlichen Einblick in den Aufbau des Rekursionsbaums, der das Programmverhalten erklären kann und durch den Programmierer von Hand kaum zu gewinnen wäre. Beispiele für solche Ausgaben finden sich in der Einleitung und bei der Beschreibung der Benchmarks in Kapitel 6.

5.4.6 Implementierung

Die konkrete Instrumentierung wird durch das `perl`-Script `instrument_program.perl` vorgenommen, das aus ca. 2 100 Zeilen Code mit 67kB Länge besteht. Es realisiert die Entwurfsalternative „eigenständiges Programm basierend auf Syntax und Mustern“ — Prozeduren werden anhand ihrer Signatur textuell erkannt, und die Instrumentierung findet ohne semantische Analyse statt. Dieser Ansatz stellt gewisse Anforderungen an das Format des Quellcodes, z.B. daß der Kopf einer Prozedur in der selben Zeile wie ihr Typ stehen muß. Sie sind in Abschnitt 5.7 aufgeführt und werden von üblichen C-Programmen bereits erfüllt.

Die Instrumentierung hat mehrere Optionen: Das Voranschreiten der Instrumentierung kann in verschiedenen Detailstufen mitverfolgt werden, die Baumaufzeichnung oder eine zeitvermessene Baumaufzeichnung können angefordert werden, und die Datensammlung kann bis auf die Erzeugung der Tiefen- und Verzweigungszähler ganz ausgeschaltet werden. Außerdem lassen sich der maximal erwartete Verzweigungsgrad und die maximal erwartete Rekursionstiefe angeben, da die entsprechenden Tabellen aus Leistungsgründen statisch aufgebaut werden. Die Vorgabewerte (Tiefe ≤ 90, Grad ≤ 12) reichen für alle betrachteten Benchmarks aus.

5.5 Parallelisierungskomponente

Wie die Instrumentierung so muß auch die Parallelisierung den Programmcode in geeigneter Form erweitern, also eine Quellcodetransformation durchführen.

5.5.1 Prinzip

Für alle zu parallelisierenden rekursiven Prozeduren werden Hüllprozeduren (*wrapper*) eingeführt, die die Prozedurparameter einkapseln, den Threadaufruf erzeugen, innerhalb des Threads die eigentliche Prozedur aufrufen und die erzeugten Threads wieder zusammenführen, sobald die Ergebnisse eines Threads benötigt werden oder das Prozedurende erreicht ist. Durch diese Hilfsprozeduren bleibt der Aufruf der Prozedur als Thread transparent. Der neue Thread bearbeitet „seinen“ Unterbaum des Problems.

Insgesamt werden für jede zu parallelisierende rekursive Prozedur P die folgenden Ergänzungen im Quellcode vorgenommen:

- Eine Datenstruktur (`struct`) mit den Parametern der Prozedur, da ein Thread als Argument nur einen einzigen Zeiger akzeptiert.

- Eine Prozedur `Thread_`P, die die Parameter von P in die Datenstruktur einpackt, einen neuen Thread mit der Hüllprozedur von P (s.u.) und einem Zeiger auf die Datenstruktur aufruft und die Threadkennung zurückgibt.
- Eine Hüllprozedur `ThreadWrapper_`P, welche die Argumente aus der übergebenen Datenstruktur auspackt und die normale sequentielle Prozedur P aufruft.
- Eine Prozedur `Join_`P, die die gegebene Threadkennung aufsammelt und garantiert, daß die Berechnung des Threads danach abgeschlossen ist.

Außerdem wird die Parallelisierungsstrategie in Form eines Prädikats am Ort der rekursiven Verzweigung eingebaut, das zur Laufzeit überprüft, ob die Strategiebedingung erfüllt ist, und ggf. einen Thread statt eines rekursiven Aufrufs erzeugt. Ein Feld von Threadkennungen zeichnet die erzeugten Threads auf, um später ihr Einsammeln zu ermöglichen.

Abbildung 5.5 verdeutlicht diese Zusammenhänge.

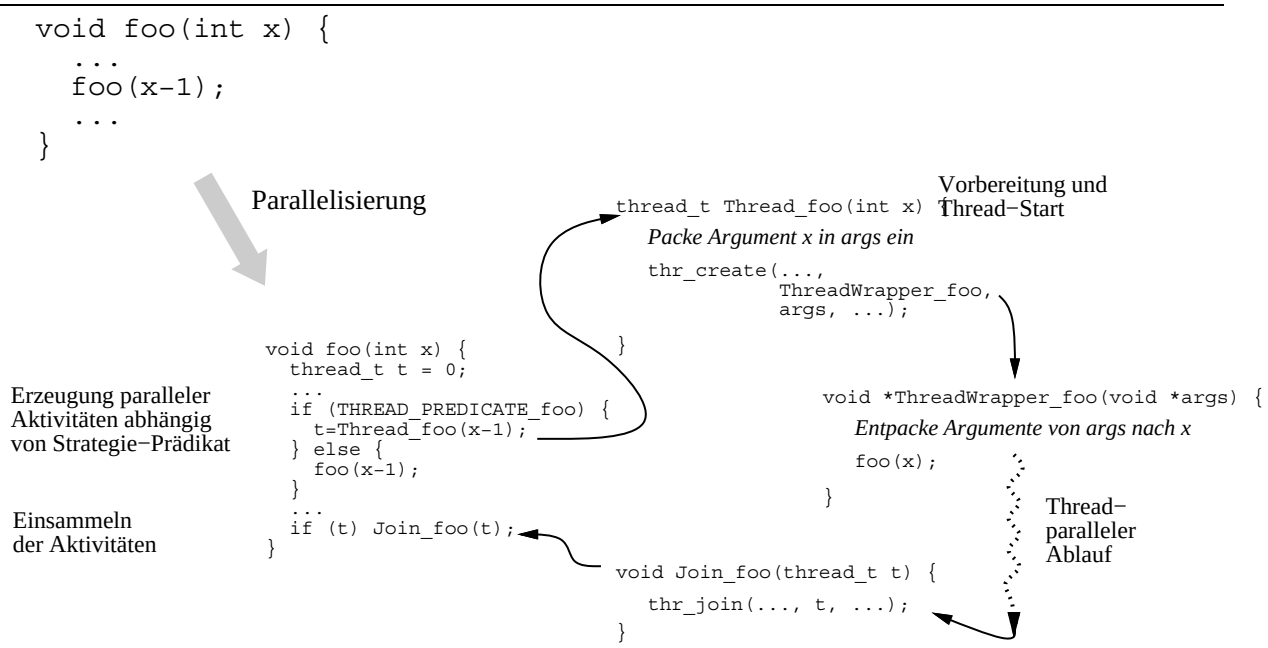


Abbildung 5.5: Ablauf der Threaderzeugung durch automatisch eingefügte Hilfsprozeduren, vereinfacht.

Die Zahl der aktuell aktiven Threads und der insgesamt erzeugten Threads wird bei Parallelisierungsstrategien, die diese Angabe benötigen, in einer durch einen kritischen Abschnitt (*r/w lock*) gesicherten Variablen mitgeführt.

5.5.2 Algorithmus

Abbildung 5.6 zeigt den Algorithmus, der aus dem gegebenen Quellcode ein threadparalleles Programm erzeugt.

Die Komplexität der Parallelisierung ist $O(R*P*C+L)$, wobei R die Zahl aller rekursiven Prozeduren, P die Zahl der parallelisierten rekursiven Prozeduren, C die Zahl der Aufrufe an parallele rekursive Prozeduren und L die Länge des Programms ist. Außerdem wird einmalig am Ende der Parallelisierung die Instrumentierung aufgerufen, um die Verzweigungszähler

```
insert general thread initializations and declarations at beginning of program code
topcode = initialization prefix
initcode = global thread initialization procedure template
for all recursive procedures R
     threadcode = thread wrapper and join procedures template for R
     if ( (R is not annotated NOPARALLEL) and (R has return type "void") )
          add defines, struct for parameters, thread wrapper, thread generation predicates,
          and procedure declaration to topcode
          if (R has no forward declaration)
               add forward procedure declaration to topcode
          end if
          add strategy depth print statement to initcode
          for all parameter variables V of R
               add thread argument initialization for V to threadcode
          end for
          insert actual procedure name, type, arguments etc. of R into threadcode
          insert threadcode before start of R
          for all recursive procedures P
               if ( (R in directcalls[P]) and (P in allcalls[R]) )
                    callcode = recursion-as-thread call code template including profile depth
                    if (P has no thread initialization code yet)
                         arrayinitcode = thread array initialization template
                         arraydecl = ""
                         joincode = thread joining code template
                         for all recursive procedures Q called in P
                              add thread array declaration for Q to arraydecl
                              add thread array initialization for Q to arrayinitcode
                         end for
                         insert arraydecl before Q's variable declaration
                         insert arrayinitcode before Q
                         for all join lines J (annotated or return) in P
                              add joincode before J
                         end for
                    end if
                    insert actual procedure name, type, arguments etc. of R into callcode
                    for all calls C of R in P
                         insert callcode around C
                    end for
               end if
          end for
     else
          annotate R as NOPARALLEL
     end if
end for
insert topcode at start of program
insert initcode before start of main()
insert call to initcode in main()
perform insertion changes
```

Abbildung 5.6: Algorithmus: Einfügen von Thread-Konstrukten zur Parallelisierung. Eingabe ist der Quellcode und Ergebnisse seiner Analyse, z.B. `allcalls[]` mit der transitiven Hülle aller Aufrufe einer Prozedur und `directcalls[]` mit den direkt in einer Prozedur durchgeführten Aufrufen. Ausgabe ist der threadparallele Quellcode.

und Tiefenparameter in den Quellcode einzufügen. Zeitmessungen hierzu finden sich im Abschnitt 6.2.9.

5.5.3 Implementierung

Die Parallelisierung erfolgt durch das `perl`-Script `parallelize_program.perl`, das ca. 2 500 Zeilen mit 82kB umfaßt. Die für die Threadfelder nötigen Zähler des Verzweigungsgrads und die Rekursionstiefenzähler für die Tiefenstrategie werden durch einen eingebetteten Aufruf von `instrument_program.perl` generiert.

Optionen des Scripts sind der Detailgrad der Ausgabe, das Abschalten der Datensammlung und die Angabe von Verzweigungsgrad und Tiefe, die an die Instrumentierung durchgereicht werden.

Sowohl Instrumentierung als auch Parallelisierung verwenden einige besondere Operationen in C: Die Ausgabe der Ergebnisse am Ende des Programms wird z.B. durch eine Definition eines `atexit()`-Handlers erreicht, der automatisch am Programmende aufgerufen wird und die Ausgaben erzeugt. Um die nötigen Datenstrukturen wie das Feld der erzeugten Threads zu initialisieren, werden Hilfsprozeduren eingeführt, die über eine Dummy-Deklaration im Variablendeklarationsteil der Prozedur aufgerufen werden — eine Plazierung direkt hinter den Deklarationen würde hingegen eine semantische Programmanalyse zur Abgrenzung des Deklarationsteils erfordern. Durch diese Hilfsmittel konnte der Aufwand der Codeänderungen geringer gehalten werden.

Alle Programmtransformationen sind so ausgelegt, daß die Lesbarkeit des Quellcodes durch deskriptive Namen und Erhalt der Einrückung gewahrt bleibt.

5.6 Strategiewahlkomponente

Im Entwurfskapitel wurde bereits erwähnt, daß REAPAR für die Strategieauswahl eine Kombination aus Heuristiken und Simulation verwendet. Die Struktur der Strategiewahlkomponente ist in Abbildung 5.7 gezeigt:

Auf oberster Ebene wird das untersuchte Programm zuerst als grob- oder feingranular klassifiziert. Feingranulare Programme können nur mit der Tiefenstrategie effizient parallelisiert werden, und eine komplette Baumaufzeichnung hätte ein zu großes Datenaufkommen. Daher entscheidet eine Heuristik aufgrund einer Analyse des Laufzeitprofils, welche Tiefenstrategie verwendet werden soll.

Grobgranulare Programme hingegen erlauben eine Baumaufzeichnung. Im Rekursionsbaum wird dann eine Simulation des Threadverhaltens durchgeführt, und die vielversprechendste Tiefen- oder allgemeine Strategie wird für die Parallelisierung empfohlen.

Die folgenden Abschnitte beleuchten die Unterkomponenten der Strategiewahl genauer.

5.6.1 Datenanalyse und Strategiewahl

Basis der Granularitätsklassifikation ist das aufgezeichnete Laufzeitprofil. Aus ihm läßt sich direkt ablesen, wieviele Blätter jede rekursive Prozedur insgesamt hatte und wie hoch die Laufzeit des Programms war. Der Wert *Laufzeit/Blattzahl* gibt die Granularität jeder Prozedur an, wobei vereinfachend angenommen wird, daß alle Blätter eine gleich hohe Laufzeit haben.

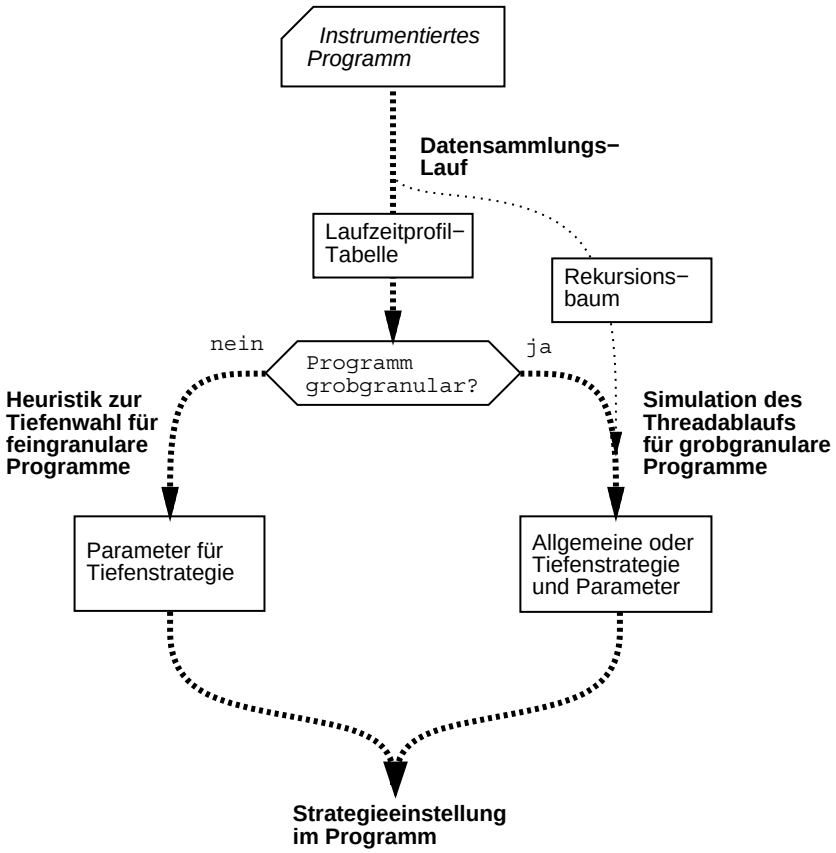


Abbildung 5.7: Entscheidungstruktur und Abläufe der Strategiewahlkomponente

Übersteigt die Laufzeit pro Blatt einen maschinenspezifischen Grenzwert, der empirisch aus den Threaderzeugungskosten und der Rechenleistung eines Einzelprozessors bestimmt wird, so gilt das Problem als grobgranular. Auf dem verwendten Vierprozessorrechner ist diese Laufzeit z.B. 1ms.

Grobgranulare Probleme ermöglichen eine Aufzeichnung ihres Rekursionsbaums mit nachfolgender Simulation des Threadverhaltens, wie sie in Abschnitt 5.6.3 beschrieben ist. Eine Strategiewahl für feingranulare Probleme kann hingegen nur aufgrund ihres tabellarischen Laufzeitprofils erfolgen, wie in den folgenden Abschnitten beschrieben wird.

5.6.2 Profilauswertung

Aufgabe der Profilauswertung ist es, nur anhand der Profiltabelle des Programms eine erfolgversprechende Rekursionstiefe für die Tiefen-Parallelisierungsstrategie zu ermitteln.

5.6.2.1 Prinzip

Allgemein gilt bei der Tiefenstrategie mit Parameter T, daß soviele Threads erzeugt werden, wie es Knoten und Blätter in den Tiefen $0 \ldots T$ gibt, weil jede rekursive Verzweigung als Thread gestartet wird (im Falle einer Nothread-Annotation kann die Threadzahl darunter liegen, aber die Heuristiken müssen den schlimmsten Fall der vollen Threaderzeugung berücksichtigen). Unterhalb der Tiefe T werden die Unterbäume sequentiell abgearbeitet.

Die Basis der Tiefenwahl sind daher die folgenden Heuristiken, wobei C für die Zahl der CPUs steht:

- Der größte Unterbaum, der sequentiell von einem Thread ausgeführt wird, darf nicht mehr als $1/C$ des Gesamtbaums umfassen. Ansonsten kann das Problem nicht gleichmäßig auf die C CPUs verteilt werden.
- Die Zahl der Knoten bis einschließlich der untersuchten Tiefe T muß mindestens C sein, aus gleichen Gründen wie bei der ersten Heuristik.
- Die Gesamtzahl der durch die Strategie erzeugten Threads, also die Zahl der Knoten bis zur Tiefe T, darf nicht größer als ca. 3 000 sein, da das Programm unter Solaris sonst abbricht.

Die Idee der Strategiewahl ist nun, für jede mögliche Tiefe T aus dem reinen Laufzeitprofil zu ermitteln, ob die obigen Heuristiken erfüllt sein können. Dazu werden mögliche Unterbäume errechnet, die unterhalb von T sequentiell ausgeführt werden, und ihre Blatt/Knotenzahlen bewertet. Die Tiefe der ersten Ebene, die die Heuristiken erfüllt, wird als Strategieparameter gewählt.

5.6.2.2 Algorithmus

Um die Blatt/Knotenzahlen eines Unterbaums, der auf Tiefe T beginnt, zu ermitteln, bietet REAPAR zwei Verfahren:

LPS (Largest Possible Subtree): Der größtmögliche Unterbaum auf Ebene T, LPS(T), beschreibt, wieviele Knoten aufgrund des Profils maximal in einem Unterbaum enthalten sein können. Die Zahl der Knoten im LPS(T) ergibt sich, indem man startend auf Tiefe T einen Knoten mit möglichst großem Verzweigungsgrad V wählt und dann auf der folgenden Ebene als seine Kinder V Knoten mit wiederum größtmöglichen Verzweigungsgrad wählt, die in der nächsten Ebene wieder möglichst hochgradig verzweigte Kinder haben usw.

Der LPS(T) wird implizit durch folgenden Algorithmus konstruiert — $p(t, v)$ ist die Anzahl von Verzweigungen mit Grad v auf Tiefe t, also das Profil. T_{max} ist die maximale Rekursionstiefe, V_{max} der maximale Verzweigungsgrad. Ausgabe des Algorithmus ist die Zahl der Knoten $LPSnodes$ im LPS(T):

```
LPSnodes = 1; getnodes = 1
for t = T to T_max
   getnextnodes = 0 /* How many nodes to get on the next layer */
   /* Get maximum number of nodes with maximum branching degree */
   for v = V_max downto 0
      n = min(getnodes, p(t,v))
      getnodes = getnodes - n
      LPSnodes = LPSnodes + n
      getnextnodes = getnextnodes + v * n
   end for
   getnodes = getnextnodes
end for
```

Nach Ende der Schleifen kann der Anteil der Knoten im LPS am Gesamtbaum mit $1/C$ verglichen werden.

AS (Average Subtree): Der durchschnittliche Unterbaum AS(T) gibt an, wie groß ein Unterbaum auf Ebene T bei durchschnittlichem Verzweigungsgrad und durchschnittlicher Knotenzahl auf jeder Ebene wird. Seine implizite Konstruktion verläuft analog zu der des LPS, nur daß statt Knoten mit möglichst großem Verzweigungsgrad hier Knoten verwendet werden, deren Verzweigungsgrad dem Durchschnitt der Verzweigungsgrade der aktuellen Baumebene entspricht. Die Zahl der Blätter *ASnodes* im AS(T) bestimmt sich durch den folgenden Algorithmus:

```
ASnodes = 1; getnodes = 1
for t = T to T_max
   /* Compute average branching degree, weighted by number of nodes */
   layernodes = 0
   avgdegree = 0
   for v = V_max downto 1
      avgdegree = avgdegree + v * p(t, v)
      layernodes = layernodes + p(t, v)
   end for
   avgdegree = avgdegree / layernodes
   n = min(getnodes, layernodes)
   nodes = nodes + n
   /* Calculate average number of nodes to get in next layer */
   getnodes = avgdegree * n
end for
```

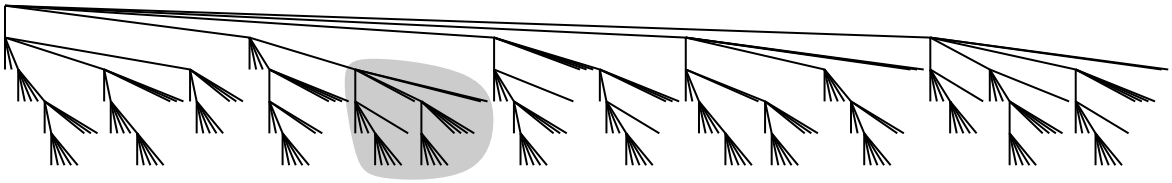
Der LPS ist die Abschätzung der Baumgröße für den schlimmsten Fall. Eine Strategie-Tiefe, die basierend auf dem LPS geschätzt wird, ist in der Realität oft zu groß. Die Werte, die eine Abschätzung mit dem AS liefert, entsprechen eher den realen Gegebenheiten. Wenn ein vollständiger Baum vorliegt, sind LPS und AS identisch, da der maximale Verzweigungsgrad immer ausgenutzt wird, also gleich dem durchschnittlichen Verzweigungsgrad ist.

Verwandt mit dem LPS ist der Begriff des *Worst Case Tree* (WCT). Der WCT eines Profils wird aufgebaut, indem für jede Ebene die Knoten beginnend mit den maximalen Verzweigungsgrad in den Baum eingefügt werden. Der linkeste Unterbaum hat also immer einen Knoten mit maximalem Verzweigungsgrad. Er stellt den LPS der jeweiligen Ebene dar. Abbildung 5.8 vergleicht zur Verdeutlichung den realen Rekursionsbaum des Queens-Benchmarks mit Eingabegröße 5 mit dem aus dem Laufzeitprofil rekonstruierten WCT. Hervorgehoben ist ein realer Unterbaum auf Ebene 2 und sein LPS-Gegenstück. Man sieht, daß der LPS deutlich größer ist als sein reales Vorbild — die Abschätzung durch einen LPS ist also konservativ.

Für Programme mit hierarchischen rekursiven Prozeduren kann REAPAR die Profile der Prozeduren zusammenfassen. Dies geschieht einfach durch Addition der Profile aller an der Rekursion beteiligten Prozeduren (das Profil einer Prozedur in unterer Hierarchieebene beginnt nicht auf Tiefe Null, sondern in der Tiefe, wo die Prozedur zum erstenmal durch die übergeordnete Prozedur aufgerufen wurde).

Auch Programme mit mehreren Iterationen eines rekursiven Aufrufs sind durch die Heuristik abgedeckt — die Zahl der Iterationen I entspricht genau der Zahl der Knoten in

Realer Rekursionsbaum mit typischem Unterbaum auf Tiefe 2



Aus Laufzeitprofil rekonstruierter "Worst Case Tree" mit größtmöglichem Unterbaum

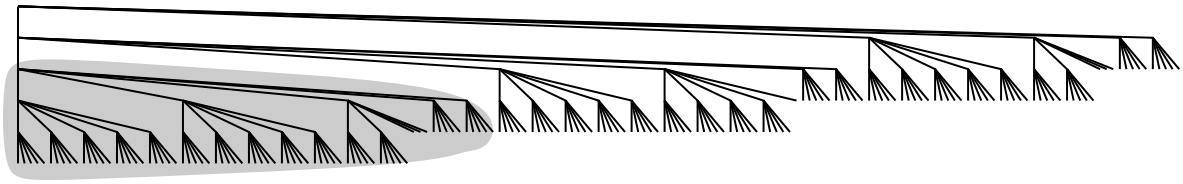


Abbildung 5.8: Realer Rekursionsbaum des Queens-Benchmarks und zugehöriger aus dem reinen Verzweigungsprofil rekonstruierter WCT. Zum Vergleich sind ein typischer Unterbaum im realen Baum und der entsprechende größtmögliche Unterbaum (LPS) im rekonstruierten Baum hervorgehoben.

Rekursionstiefe Null. Die Profilauswertung erfolgt dann in einem Profil, bei dem alle Einträge durch I geteilt wurden, also basierend auf dem Durchschnitt aller Rekursionsbäume der Prozedur.

5.6.2.3 Beispielausgabe

Abbildung 5.9 zeigt ein typisches tabellarisches Laufzeitprofil, wie es durch die Instrumentierung automatisch erzeugt wird, am Beispiel des Fractal-Benchmarks. Der darunter abgebildete Lauf der AS-Heuristik stellt fest, daß Tiefe 2 ausreicht, um durchschnittliche Unterbäume auf die 4 Prozessoren zu verteilen. Wie sich in Kapitel 6 herausstellt, ist Depth 2 in der Tat die beste Strategie zur Parallelisierung dieses Problems. Außerdem empfiehlt die Auswertung, eine leistungssteigernde Nothread-Annotation zu verwenden, da der Verzweigungsgrad immer 0 oder 4 ist.

5.6.2.4 Implementierung

Die Tiefenwahl ist als C-Programm namens `evaluate_profile` mit 800 LOC in 24kB Quelltext implementiert. Sie liest das Laufzeitprofil von der Standardeingabe und führt die Strategiewahl je nach Option basierend auf dem LPS oder AS durch.

Da die Bäume nur implizit durchlaufen, aber nicht wirklich im Speicher aufgebaut werden, läuft die Heuristik sehr schnell ab, wie Abschnitt 6.2.10 zeigt.

Eine Stufe darüber liegt das Script `choose_strategy_by_heuristics`, das für ein gegebenes Laufzeitprofil und eine CPU-Zahl die Heuristikauswahl aufruft und die erste gewählte Tiefe ausgibt bzw. darauf hinweist, wenn die Heuristik keine Tiefe als vorteilhaft beurteilen konnte. Optional kann statt der vorgegebenen AS-Heuristik auch die LPS-Heuristik verwendet werden.

```
  226 - compute_area
        (  0:     0    0    0    0     1    0    0  sum =     1)
        (  1:     0    0    0    0     4    0    0  sum =     4)
        (  2:     0    0    0    0    16    0    0  sum =    16)
        (  3:     8    0    0    0    56    0    0  sum =    64)
        (  4:    50    0    0    0   174    0    0  sum =   224)
        (  5:   177    0    0    0   519    0    0  sum =   696)
        (  6:   633    0    0    0  1443    0    0  sum =  2076)
        (  7:  1801    0    0    0  3971    0    0  sum =  5772)
        (  8:  4891    0    0    0 10993    0    0  sum = 15884)
        (  9: 43972    0    0    0     0    0    0  sum = 43972)
        ( 10:     0    0    0    0     0    0    0  sum =     0)
        (sum: 51532    0    0    0 17177    0    0  SUM = 68709)

> ./evaluate_profile -a < profile_fractal1_1024_1024
profiled procedure 'compute_area' at source line 226
maximum degree = 4, maximum depth = 9
profile covers 1 iterations
recommend using the NOTHREAD annotation for branching degree 4
 since degree is always 0 or 4.
performing Average Subtree analysis for 5 CPUs

analyzing depth 1:

all in all, the average subtree has 17178.0 nodes.
that's  25.00% of the tree's nodes
-> not recommended for 5 CPUs, average subtree not smaller than 1/5 (20.00%)

analyzing depth 2:

all in all, the average subtree has 4295.0 nodes.
that's   6.25% of the tree's nodes
-> RECOMMENDED: depth  2 for  5 CPUs, average subtree is only 6.25%
```

Abbildung 5.9: Beispielhaftes Laufzeitprofil des Fractal-Benchmarks mit Eingabegröße 1024^2 und entsprechender Lauf der Auswertungsheuristik mit AS für 5 CPUs.

5.6.3 Simulation

Für den Fall eines grobgranularen Problems wird der komplette Rekursionsbaum aufgezeichnet. Er spiegelt den exakten Ablauf des Programms wider, im Gegensatz zu dem mit Informationsverlusten behafteten Laufzeitprofil. Daher kann er als Grundlage einer Simulation dienen, die das parallele Ablaufverhalten des Programms für beliebige Parallelisierungsstrategien nachbildet.

5.6.3.1 Prinzip

Aufgabe der Simulation ist es, anhand eines aufgezeichneten Rekursionsbaums R den threadparallelen Ablauf des Programms auf C Prozessoren unter Verwendung einer Parallelisierungsstrategie S zu simulieren: Der Baum R wird aus einem Laufzeitprofil mit Baumaufzeichnung gewonnen. Er enthält die genauen Informationen, welche Prozedur an welcher Stelle der Rekursion welche andere aufgerufen hat. Basierend auf der jeweiligen Paralleli-

sierungsstrategie werden während der Simulation an den Knoten des Baums virtuelle neue Threads erzeugt. Die Threads werden in einer Warteschlange gehalten, aus der sich die C virtuellen Prozessoren jeweils einen ablaufbereiten Thread nehmen und solange bearbeiten, bis er keine Knoten mehr zu besuchen hat oder er auf das Ende von erzeugten Kind-Threads wartet. Der genaue Algorithmus ist im nächsten Abschnitt beschrieben.

Folgende Vereinfachungen werden bei der Simulation gemacht:

- Die Ausführung jedes Knotens dauert genau einen Zeitschritt.
- Erzeugung, Wechsel und Beenden von Threads geschehen verzögerungsfrei in Null Zeitschritten.
- Ein Prozessor bearbeitet einen Thread solange, bis dieser keine Arbeit mehr hat oder auf seine Kinder wartet (d.h. eventuelle Threadwechsel durch Zeitscheibenablauf des Vaterprozesses werden nicht simuliert).
- Da aus dem Baum keine Threadzusammenführungsstellen hervorgehen, wird pro Knoten einmalig direkt vor der Rückkehr zum aufrufenden Knoten auf alle erzeugten Threads gewartet.

Das Ergebnis einer Simulation ist der Rekursionsbaum mit Annotationen, welcher Thread welche Knoten ausgeführt hat, und die Zahl der benötigten Simulationsschritte. Abgeleitete Aussagen sind die simulierte Auslastung der Maschine, die Größe der Arbeitseinheiten als Zahl der Knoten, die ein Thread bearbeitet hat, und verschiedene Statistiken über diese Größen, die später anhand eines Beispiels genauer vorgestellt werden.

Für Programme mit mehreren Iterationen wird wie eingangs erwähnt pro Iteration (d.h. Aufruf der Rekursion von außerhalb) ein Rekursionsbaum aufgezeichnet. Die Simulation findet in diesem Fall nur für den ersten Rekursionsbaum statt — denkbar wäre alternativ eine Simulation für mehrere der Rekursionsbäume und die Mittelwertbildung der Strategien oder die Wahl der am häufigsten einzeln gewählten Strategie als Gesamtstrategie. Wie sich in der Auswertung in Abschnitt 6.2.5 herausstellt, ist aber eine Strategie auch für einen recht weiten Bereich verwandter Probleme gültig, also insbesondere auch für andere Iterationen desselben Programms. In der Praxis reicht daher die Simulation der ersten Iteration aus.

Trotz der teilweise starken Vereinfachungen liefert die Simulation sehr realitätsnahe Aussagen über den threadparallelen Ablauf des Programms. Sogar die Zahl der Simulationsschritte korrespondiert gut mit der realen Laufzeit, d.h. Strategien, deren Simulation weniger Schritte benötigt, sind auch fast immer real schneller.

Nach Abschluß der Simulation wird durch Heuristiken beurteilt, wie gut die simulierte Strategie zur Parallelisierung des Problems geeignet ist. Der nächste Abschnitt beschreibt den Algorithmus der Simulation und diese Heuristiken.

5.6.3.2 Algorithmus

Die Simulation basiert auf virtuellen Threads und Prozessoren, die die Threads im Rekursionsbaum ausführen. Folgende Datenstrukturen werden verwendet:

Ein simulierter Thread T im Thread-Feld `threads[]` befindet sich in einem der Zustände NONE (inaktiv), ACTIVE (aktuell an einer Berechnung beteiligt), WAITING (Warten auf Kind-Threads), DONE (fertig mit der Berechnung) oder JOINED (nach Berechnung vom Eltern-Thread aufgesammelt). Der simulierte Thread enthält Informationen über seinen

Zustand, die Zahl der Knoten, die er bearbeitet hat, die Zahl von bearbeiteten nicht parallelisierten Knoten, seine Start- und Maximalrekursionstiefe, seinen Startknoten und seinen aktuellen Knoten.

Eine simulierte CPU C hat einen aktuell auf ihr ablaufenden Thread (`cputhreads[`C`]`) oder ist als „untätig“ markiert. Die Untätigkeit der CPUs wird am Ende der Simulation benutzt, um die simulierte Last der Maschine zu bestimmen.

Knoten K im Baum kennen ihre Kinder, ihren Elternknoten, die Zahl von Knoten in ihrem Unterbaum, die sie erzeugende Prozedur, den sie bearbeitenden Thread und ihre Tiefe im Baum. Außerdem zeigen Wahrheitsvariablen an, ob der Knoten fertig bearbeitet wurde (Zustand DONE) und ob alle Unterknoten von nichtparallelen Prozeduren erzeugt wurden, der Knoten also ein „paralleles Blatt“ ist.

Weiterhin werden für die Prozeduren P, die den Rekursionsbaum aufgebaut haben, Information gehalten, ob sie parallelisiert sind, ob und für welchen Verzweigungsgrad eine Nothread-Annotation galt, wieviele Knoten sie erzeugten und wieviele nichtparallele Unterknoten sie indirekt generierten.

Abbildung 5.10 stellt vor diesem Hintergrund den Kernalgorithmus der Simulation dar, ohne die Details wie Aktivitätszähler oder Blattzähler zu berücksichtigen. Im realisierten Algorithmus wird außerdem optional die Abarbeitung von sequentiellen Unterbäumen simuliert, die durch nichtparallelisierte Hierarchieebenen entstehen (`parnodes` und `parleafs` im Beispiel 5.11, d.h. die oben angesprochenen Knoten/Blätter, die das Ende der parallelen Ausführung bedeuten, aber sequentiell weiterführen).

Die Heuristiken, die nach Abschluß der Simulation die Parallelisierung beurteilen, sind für C CPUs:

- Es wurden mindestens C Threads erzeugt.
- Die Zahl der Knoten bzw. Blätter im größten durch einen einzigen Thread abgearbeiteten Unterbaum ist maximal $1/C$ des Gesamtbaums.
- Gleiches gilt für die Zahl der Knoten und Blätter, die von sequentiell abgearbeiten rekursiven Prozeduren stammen (z.B. wenn nur die oberste Rekursionshierarchie parallelisiert wurde).
- Jede simulierte CPU ist mindestens 75% der Zeit ausgelastet

Die ersten drei Punkte sind notwendig für eine effiziente Parallelisierung, die CPU-Auslastung hingegen kann in der Simulation unter dem Wert der Heuristik liegen und real doch hinreichend groß sein.

5.6.3.3 Implementierung

Die Simulation ist aus Leistungsgründen in C geschrieben (ca. 1 700 Zeilen und 53kByte Quellcode).

Als Optionen nimmt sie die zu simulierende Strategie, die Zahl der Prozessoren und die Einstellung der Menge an ausgegebener Information an, bis hin zum Ausdruck des Rekursionsbaums mit den aktuell aktiven Threads nach jedem Simulationsschritt. Außerdem läßt sich angeben, welche Prozeduren als parallel simuliert werden sollen, ob und welche Verzweigung als Nothread annotiert wurde, und ob die Unterbäume nichtparalleler Prozeduren in einem einzigen Simulationsschritt oder sequentiell abgearbeitet werden sollen.

```
read the recursion tree and parse it, noting which procedures executed which nodes
parse command line parameters, e.g. strategy and number of processors
initialize the virtual threads and the thread queue
for all C in CPUs:  cputhreads[C] = NONE
thread[0].node = rootnode; cputhreads[0] = 0; simsteps = 0
while rootnode.state != DONE                                   Simulate until whole tree done
     for all C in CPUs
          childrendone = False                                 Are all the node's children done?
          T = cputhreads[C]                                    Get current thread for this CPU
          if T is NONE
               T = dequeue_thread(C)                           CPU has no current thread, dequeue one
               if T is NONE then
                    Mark CPU idle and skip to next CPU
               end if
          end if
          N = T.node
          if all of N's children are DONE, JOINing them while checking
               N.state = DONE                                 All children done → node done
               if T.startnode == N                            At thread's start node? Thread done!
                    cputhreads[C] = NONE                      Remove thread from CPU
               T.state = DONE; break for
               else                                           Otherwise: Proceed upwards in tree
                    T.state = ACTIVE
                    T.node = N.parent
               end if
               childrendone = True
          end if
          switch (T.state)
          case WAITING:
               if not childrendone                            Still waiting for children, thread sleeps
                    cputhreads[C] = NONE                      Remove thread from CPU...
                    enqueue_thread(T)                         ...and put it in Q for later use
               end if
          case ACTIVE:
               if not childrendone                            Children left to work on?
                    if all of N's children are busy
                         T.state = WAITING
                    else                                      Start work on child as new thread...
                         if parallelization strategy says so
                              spawn_and_enqueue_new_thread(N.child)
                         else                                 ...or compute child sequentially
                              T.node = N.child
                         end if
                    end if
               end if
          end switch
     end for
     simsteps++
end while
```

Abbildung 5.10: Algorithmus: Simulation des Threadablaufs im Rekursionsbaum. Eingabe ist der Rekursionsbaum und die Parameter der Simulation, Ausgabe der mit Threadinformationen annotierte Rekursionsbaum und abgeleitete Werte wie die Zahl der Knoten im größten Thread.

Die Simulation erkennt auch, wenn mehr als 3 000 Threads gleichzeitig aktiv sind, und bricht mit einer Warnung ab. Das verhindert, daß Strategien empfohlen werden, die später unter Solaris zum Programmabbruch führen würden.

5.6.3.4 Beispiel

Die Ergebnisse einer Simulation des parallelen Ablaufs des Barnes Hut Benchmarks mit Eingabegröße 8 auf 4 Prozessoren zeigt der Ausdruck in Abbildung 5.11. Die ASCII-Baumausgabe eines Zwischenzustands der Simulation ist in Abbildung 5.12 zu finden.

5.6.3.5 Strategiewahl

Eine einzelne Simulation gibt nur Hinweise, wie gut die aktuell simulierte Strategie abschneidet. Für die globale Strategiewahl müssen verschiedene Strategien simuliert und ihre Ergebnisse verglichen werden. Dafür gibt es zwei Möglichkeiten:

- Simulation aller eventuell sinnvollen Strategien, z.B. Depth 1...20, Keep 1...15 und Active 1...15, und nachfolgende Auswahl der Strategie, die die geringste Zahl von Simulationsschritten benötigte.[2]
- Simulation von jeweils Depth, Keep und Active mit zunehmendem Strategieparameter, aber Abbruch der Simulationsläufe, sobald die Heuristiken der gerade simulierten Strategie eine gute Parallelisierung versprechen.

Es stellt sich heraus, daß beide Methoden in der Praxis gleich gute Schätzungen liefern, was für die Qualität der Simulation spricht. Das zweite Verfahren läuft naturgemäß schneller ab, da es weniger Strategien zu simulieren hat.

Realisiert wird diese Ebene der Strategiewahl durch das Skript `choose_strategy_by_simulation`, das für einen gegebenen Rekursionsbaum und eine CPU-Zahl Simulationsläufe für jede der drei Strategieklassen startet und den Strategieparameter erhöht, bis die Filterheuristiken der Simulation die Strategie empfehlen oder ein Grenzwert des Parameters überschritten wurde. Die dadurch gewählten Strategien werden dann nach der Zahl ihrer jeweiligen Simulationsschritte aufsteigend sortiert und ausgegeben. Strategieklassen, für die kein befriedigender Parameter gefunden wurde, werden als solche markiert und zuunterst eingeordnet. Als Optionen nimmt das Skript die oben erwähnte Baumbehandlung der Simulation an und reicht sie an diese weiter, z.B. um eine Nothread-Annotation nachzubilden oder wenn untere Rekursionshierarchien nicht als parallel simuliert werden sollen. Die sortierte Endausgabe sieht z.B. wie folgt aus:

```
> choose_strategy_by_simulation 8 recorded_eigenvalue_g_1500  '-b A1'

Overall ranking:
Rank 1: Active  3  (607 steps)
Rank 2: Keep   17  (634 steps)
Rank 3: [Depth not recommended]
```

Die erstplazierte Strategie wird von REAPAR für die Parallelisierung des Programms verwendet.

[2] Eine Binärsuche nach dem Strategieparameter ist nicht hilfreich, da die Beschleunigungskurven weder monoton noch bitonisch sind. Zudem haben gute Strategie meistens kleine Parameter.

```
> simulation -k 4 -c 4 < recorded/recorded_barnes_8
  Thread   0: start   0, max   0,   1 nodes,   0 leafs,   1 subnodes,   0 subleafs
  Thread   1: start   1, max   5,   4 nodes,   8 leafs,   4 subnodes,   8 subleafs
  Thread   2: start   2, max   2,   1 nodes,   0 leafs,   1 subnodes,   0 subleafs
  Thread   3: start   3, max   3,   1 nodes,   0 leafs,   1 subnodes,   0 subleafs
  Thread   4: start   4, max   5,   1 nodes,   4 leafs,   1 subnodes,   4 subleafs
  Thread   5: start   1, max   1,   1 nodes,   0 leafs,   1 subnodes,   0 subleafs
  Thread   6: start   2, max   2,   1 nodes,   0 leafs,   1 subnodes,   0 subleafs
  Thread   7: start   4, max   4,   0 nodes,   1 leafs,   0 subnodes,   1 subleafs
  Thread   8: start   2, max   5,   3 nodes,   8 leafs,   3 subnodes,   8 subleafs
  Thread   9: start   2, max   5,   3 nodes,   8 leafs,   3 subnodes,   8 subleafs
  Thread  10: start   5, max   5,   0 nodes,   1 leafs,   0 subnodes,   1 subleafs
  Thread  11: start   2, max   4,   2 nodes,   8 leafs,   2 subnodes,   8 subleafs
  Thread  12: start   2, max   5,   3 nodes,   6 leafs,   3 subnodes,   6 subleafs
  Thread  13: start   5, max   5,   0 nodes,   1 leafs,   0 subnodes,   1 subleafs
  Thread  14: start   3, max   5,   2 nodes,   6 leafs,   2 subnodes,   6 subleafs
  Thread  15: start   2, max   5,   3 nodes,   5 leafs,   3 subnodes,   5 subleafs
  Thread  16: start   5, max   5,   0 nodes,   1 leafs,   0 subnodes,   1 subleafs
  Thread  17: start   5, max   5,   0 nodes,   1 leafs,   0 subnodes,   1 subleafs
  Thread  18: start   5, max   5,   0 nodes,   1 leafs,   0 subnodes,   1 subleafs
  Thread  19: start   5, max   5,   0 nodes,   1 leafs,   0 subnodes,   1 subleafs
  Thread  20: start   5, max   5,   0 nodes,   1 leafs,   0 subnodes,   1 subleafs
  Thread  21: start   4, max   4,   0 nodes,   1 leafs,   0 subnodes,   1 subleafs
  Thread  22: start   4, max   4,   0 nodes,   1 leafs,   0 subnodes,   1 subleafs
  Thread  23: start   4, max   4,   0 nodes,   1 leafs,   0 subnodes,   1 subleafs

Simulation for KEEP 4 on 4 CPUs done. 23 new threads generated over all
57 simulation steps elapsed, CPU utilization = 339%
All procedures simulated as parallelized

Procedure B:    10 nodes,     0 leafs,    10 parnodes,     0 parleafs
Procedure C:    16 nodes,    64 leafs,    16 parnodes,    64 parleafs

Thread statistics for 24 threads:
 nodes:        0 min,     1 med,     4 max,     1.08 avg,    1.256 dev ( 1.159)
 leafs:        0 min,     1 med,     8 max,     2.67 avg,    2.953 dev ( 1.108)
 subnodes:     0 min,     1 med,     4 max,     1.08 avg,    1.256 dev ( 1.159)
 subleafs:     0 min,     1 med,     8 max,     2.67 avg,    2.953 dev ( 1.108)
Largest values:     4 nodes    ( 15.38%),   8 leafs    ( 12.50%),
                    4 subnodes ( 15.38%),   8 subleafs ( 12.50%)

Heuristics:
No fewer threads than CPUs                          : OK
Node percentage of largest <= 1 CPU percentage      : OK
Leaf percentage of largest <= 1 CPU percentage      : OK
Subnode percentage of largest <= 1 CPU percentage   : OK
Subleaf percentage of largest <= 1 CPU percentage   : OK
Each CPU used at least 75% of the time              : OK
```

Abbildung 5.11: Beispiel-Ausgabe einer Simulation der Strategie Keep 4 auf 4 Prozessoren mit anschließender statistischer Auswertung und Ergebnissen der Heuristik. Pro Thread wird seine Startebene, die maximal erreichte Ebene und die Zahl der von ihm bearbeiteten Knoten und Blätter und sequentiellen Unterbäume ausgegeben. Eine Auslastung von 400% bedeutet, daß alle 4 CPUs zu 100% beschäftigt waren.

```
Thread queue:  14  15  16   5   6   2   0   4   3   1
CPU 2 thread 11 - state A, depth 4, current node 0x624, start node 0x232  <-- Thread 11 wird gerade auf
        B.000W--B.001W--B.002W--C.003W--C.004W--Cd010D                        CPU 2 ausgefuehrt. Aktiv sind
                |       |               |       Cd013D                        ausserdem 12, 15 und 16 ("A")
                |       |               |       C.016A
                |       |               |       Cd004W
                |       |               |       Cd004W             Knotenformat: PznnnT
                |       |               |       Cd004W                           || | |
                |       |               |       Cd004W                           || | Zustand des Threads:
                |       |               Cd007D                                   || |  A - Active
                |       B.006W--C.014A--C.====--C.====                           || |  D - Done
                |       |               |       C.====                           || |  J - Joined
                |       |               |       C.====                           || |  W - Waiting
                |       |               |       C.====                           || |
                |       |               |       C.====                           || Nummer des den Knoten
                |       |               |       C.====                           || ausfuehrenden Threads
                |       |               |       C.====                           || oder "====" fuer noch
                |       |               C.====                                   || nicht bearbeitete
                |       Bd008D--Cd008D--Cd008D--Cd008D                           || Knoten
                |       |               |       Cd008D                           ||
                |       |               |       Cd008D                           |Zustand des Knotens:
                |       |               |       Cd008D                           | d - done
                |       |               |       Cd008D                           | . - unbearbeitet
                |       |               |       Cd008D                           |
                |       |               |       Cd008D                           Prozedur des Knotens (hier
                |       |               Cd008D                                   "B" oder "C")
                |       Bd009D--Cd009D--Cd009D--Cd009D
                |       |               |       Cd009D
                |       |               |       Cd009D
                |       |               |       Cd009D
                |       |               |       Cd009D  <---- Beispiel 1: Dieser Unterbaum wurde von Thread 9
                |       |               |       Cd009D                    bearbeitet, alle Knoten sind "done",
                |       |               |       Cd009D                    der Thread selbst ist ebenfalls im
                |       |               Cd009D                            Zustand "Done".
                |       B.012A--C.012A--C.012A--C.====
                |       |               |       C.====
                |       |               |       C.====
                |       |               |       C.====  <---- Beispiel 2: Dieser Unterbaum wird gerade von
                |       |               |       C.====                    Thread 12 bearbeitet, der "Active"
                |       |               |       C.====                    ist. Untere Knoten der Prozedur "C"
                |       |               |       C.====                    sind noch unbearbeitet.
                |       |               C.====
                |       B.015A--C.====--C.====--C.====
                |       |               |       C.====
                |       |               |       C.====
                |       |               |       C.====
                |       |               |       C.====
                |       |               |       C.====
                |       |               |       C.====
                |       |               C.====
                |       Bd001W--Cd001W--Cd001W--Cd001W
                |                       |       Cd001W
                |                       |       Cd001W
                |                       |       Cd001W
                |                       |       Cd001W
                |                       |       Cd001W
                |                       |       Cd001W
                |                       Cd001W
                |               B.005W--C.011A--C.011A--Cd011A
                |                               |       Cd011A
                |                               |       Cd011A
                |                               |       Cd011A
                |                               |       Cd011A
                |                               |       C.011A
                |                               |       C.====
                |                               C.====
```

Abbildung 5.12: Beispiel-Ausgabe eines Zwischenzustands der Simulation der Strategie Keep 4 auf 4 Prozessoren im Rekursionsbaum von Barnes Hut (manuell annotiert).

5.6.3.6 Wahl von sequentiellen Rekursionshierarchien

Die Noparallel-Annotation eröffnet, wie in Abschnitt 5.3 beschrieben, die Möglichkeit, unnötig feingranulare Hierarchien von rekursiven Prozeduren sequentiell abzuarbeiten und damit die Granularität der darüberliegenden Rekursionshierarchie zu erhöhen. Ruft etwa die Prozedur a sich selbst und b auf, und b ruft nur sich selbst auf, dann kann es sinnvoll sein, b gar nicht zu parallelisieren, sondern nur die Parallelität von a zu nutzen. Um die Leistung nicht zu mindern, darf aber der größte Teilbaum von b nicht so groß werden, daß er die Ausführung des Gesamtprogramms zu sehr sequentialisiert.

Das REAPAR Hilfsprogramm `hierarchies_check_simulation` gibt für gegebenen Rekursionsbaum und CPU-Zahl dem Benutzer Hinweise, welche Prozeduren von einer Noparallel-Annotation profitieren könnten. Dazu analysiert es die Profilausgabe, erkennt die Aufrufhierarchie und führt daraufhin Simulationen im Rekursionsbaum des Programms mit der Always-Strategie durch. Diese Simulationen zeigen das Parallelitätspotential auf. Sie werden zunächst mit simulierter Parallelisierung aller Hierarchieebenen durchgeführt und dann mit immer einer Ebene weniger, bis nur noch die oberste Hierarchieebene parallelisiert wird.

Wenn die Auswahlheuristiken aus Abschnitt 5.6.3.2 eine hinreichende Parallelisierung vorhersagen, wird empfohlen, die als nicht parallel simulierten Prozeduren mit der Noparallel-Annotation zu versehen. Abbildung 5.13 zeigt die Ausgabe des Programms für zwei Benchmarks. Die Empfehlungen resultieren in der Realität tatsächlich in einem deutlichem Leistungsgewinn, wie später in Abschnitt 6.2.6 deutlich wird.

5.7 Voraussetzungen und Einschränkungen

Ein Programm, das vom System parallelisiert werden soll, muß eine Reihe von Voraussetzungen erfüllen:

- Es muß in ANSI C geschrieben sein, besonders in Hinsicht auf Funktionsköpfe. Funktionstyp, -name und die öffnende Klammer der Parameterliste müssen in der selben Programmzeile stehen. Die weiteren Parameter können mehrere Zeilen überspannen. Es wird nur syntaktisch korrekter Quellcode verarbeitet.

  Diese Anforderungen stellen sicher, daß Funktionen und ihre Parameter durch Mustervergleiche im Quellcode gefunden werden können, wobei die verwendeten regulären Ausdrücke unempfindlich gegenüber Formatierungsdetails wie z.B. der Zahl von Leerzeichen oder Tabulatoren sind. Sie ist in üblichen Programmen bereits erfüllt.

- Zu parallelisierende Prozeduren müssen den Rückgabetyp `void` haben.

  Dies erleichtert die Umstrukturierung des Programmcodes — anderenfalls wären rekursive Aufrufe der Art `e = f(x) + f(y)` möglich, die für den Aufruf als Thread das automatische Einführen von temporären Variablen und weitere Kunstgriffe erfordern würden.

  Die Rückgabe von Ergebnissen aus rekursiven Prozeduren ist aber alternativ über die Verwendung von Zeigerparametern möglich.

- Rekursive Prozeduren dürfen keine Datenabhängigkeiten zwischen den rekursiven Aufrufen enthalten, wenn sie parallelisiert werden sollen. Prozeduren mit sequentialisierenden Datenabhängigkeiten sind als Noparallel zu annotieren.

```
Analyzing recursion tree 'recorded_barnes_16384' on 16 CPUs
for potential non-parallelization of recursion hierarchies

Recursion tree starts with procedure computesubtree (B)
computesubtree calls walksub (C)

Simulating with all 2 procedures parallel (BC)
-> Simulation predicts enough parallelism if all procedures are parallelized
   (largest sequential chunk: 0.49% of all nodes and 0.20% of all leafs)

Simulating with first procedure parallel (B)
-> Simulation predicts enough parallelism if the procedure
   'walksub' is executed sequentially
   (largest sequential chunk: 3.92% of all nodes and 5.13% of all leafs)

   -> Recommend /* NOPARALLEL */ annotation for 'walksub'



Analyzing recursion tree 'recorded_power_10_20_5_10' on 4 CPUs
for potential non-parallelization of recursion hierarchies

Recursion tree starts with procedure Compute_Feeder (A)
Compute_Feeder  calls Compute_Lateral (B)
Compute_Lateral calls Compute_Branch (C)

Simulating with all 3 procedures parallel (ABC)
-> Simulation predicts enough parallelism if all procedures are parallelized
   (largest sequential chunk: 0.11% of all nodes and 0.56% of all leafs)

Simulating with first 2 procedures parallel (AB)
-> Simulation predicts enough parallelism if the procedure
   'Compute_Branch' is executed sequentially
   (largest sequential chunk: 0.78% of all nodes and 1.13% of all leafs)

   -> Recommend /* NOPARALLEL */ annotation for 'Compute_Branch'

Simulating with first procedure parallel (A)
-> Simulation predicts enough parallelism if the procedures
   'Compute_Lateral' and 'Compute_Branch' are executed sequentially
   (largest sequential chunk: 16.46% of all nodes and 15.25% of all leafs)

   -> Recommend /* NOPARALLEL */ annotation for 'Compute_Lateral' and 'Compute_Branch'
```

Abbildung 5.13: Beispiels-Empfehlungen von REAPAR für die Verwendung der Noparallel-Annotation für die Benchmarks Barnes Hut und Power (leicht gekürzt).

In manchem Fällen lassen sich Datenabhängigkeiten durch die Einführung neuer lokaler Variablen beheben, die erst nach Abschluß der rekursiven Berechnungen nach einer Needresults-Annotation verwendet werden.

Außerdem gibt es noch einige Anforderungen an die Klammerung im Programm, die in üblichen Programmen erfüllt sind und die der technische Bericht [43] genauer erläutert. Zudem muß der relevante Quellcode, d.h. die rekursiven Prozeduren und die Stellen ihrer Verwendung, in einer einzigen Datei vorliegen.

Kapitel 6

Ergebnisse

Die Leistung des in den vorigen Kapiteln beschriebenen Systems wurde mit verschiedenen Benchmark-Programmen getestet, um die Thesen der Arbeit zu validieren. Dieses Kapitel beschreibt die Benchmarks, ihre speziellen Herausforderungen und Charakteristika, die erreichten Beschleunigungen und die Qualität der automatischen Strategiewahl. Außerdem wird eine Fallstudie vorgestellt, die die Ergebnisse des Systems mit manuell parallelisierten Programmen und dem dazu notwendigen Aufwand vergleicht.

6.1 Benchmark-Suite

Dieser Abschnitt beschreibt die verwendeten Benchmarks und ihre Kenngrößen.

Als Benchmarks herangezogen wurden Programme, die möglichst gut das erwartete Anwendungsspektrum abdecken — die Bereiche Mathematik, naturwissenschaftliche und wirtschaftliche Simulation und Grafik sind vertreten. Es wurden explizit keine der „üblichen“ kleinen Beispiele wie FFT oder Matrixmultiplikation verwendt, da diese sehr regulär sind, nur als Unterprobleme in größeren Algorithmen auftreten und in der Praxis sowieso mit speziellen hochoptimierten Bibliotheken realisiert werden.

Im folgenden werden die für die Konstruktion des Systems betrachteten Benchmarks vorgestellt und charakterisiert. Zusätzlich wird jeweils ein beispielhafter Rekursionsbaum abgebildet, der aus realen Programmläufen automatisch gewonnen wurde. Die Benchmarks der Validierungsmenge, die nicht zum Systementwurf dienten, finden sich im nächsten Kapitel.

6.1.1 Barnes Hut

Anwendung: Simulation — Für die Simulation des Verhaltens von Galaxien werden die wirkenden Anziehungskräfte berechnet, indem die Körper zunächst in einen Baum eingeordnet werden. Die Traversion des Baums zur Berechnung der Kraft auf jeden Körper hat dann nur noch einen Aufwand von $O(n \log n)$ statt $O(n^2)$ im naiven Algorithmus, wie die Arbeit von J.Barnes zeigt [5].

Algorithmus: Die Simulation findet in mehreren Zeitschritten statt. Pro Zeitschritt wird ein achtfach verzweigter Baum (*oct-tree*) aufgebaut, in den die Körper aufgrund ihrer Lage im Raum eingeordnet werden. Jeder Knoten im Baum enthält dabei Information über das Massezentrum der in seinen Unterbäumen enthaltenen Körper.

In der Berechnungsphase, die den weitaus größten Teil der Arbeit ausmacht, wird für jeden Körper die auf ihn wirkende Kraft aller anderen Körper durch einen Baumdurchlauf errechnet: Die Blätter des Baums enthalten die Körper. Für nahe Körper wird der Baum bis zu den Blättern durchlaufen und deren Kraft auf den gerade betrachteten Körper berechnet. Räumlich weiter entfernte Gruppen von Körpern (also Unterbäume) werden hingegen durch ihr Massezentrum approximiert, d.h. für entfernte Körper wird nur ihr übergeordneter Knoten im Baum besucht und nicht alle Einzelblätter. Dadurch wird der Baum pro Körper nicht n mal, sondern nur etwa $\log n$ mal durchlaufen. Abbildung 6.1 visualisiert dieses Vorgehen für den 2-dimensionalen Fall, also mit nur vierfacher Verzweigung (*quad-tree*).

Nach der Bestimmung der Kräfte werden die neuen Positionen der Körper errechnet, und die nächste Iteration beginnt.

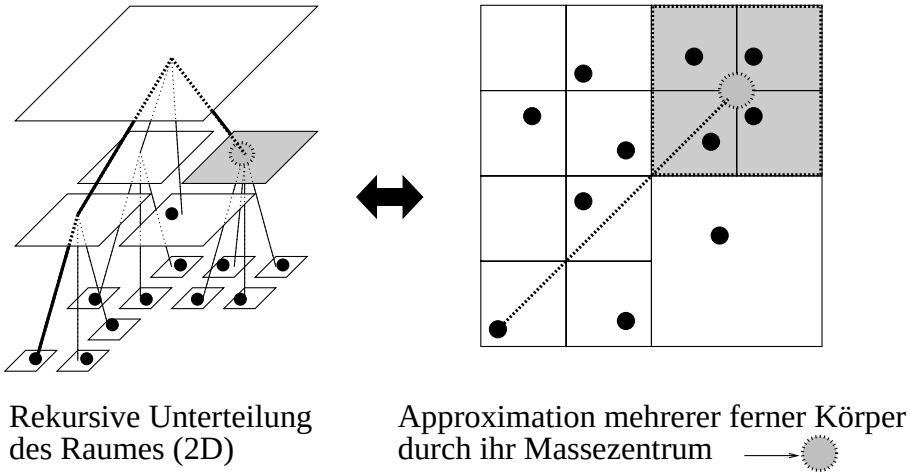


Abbildung 6.1: Barnes Hut — Einordnung der Körper in den Baum und dessen Verwendung. Die Punkte stehen für die Körper; die rekursive Unterteilung des Raums endet, wenn sich nur noch ein Körper im Quadranten befindet.

Parallelität: Die Berechnungsphase wird parallelisiert durch gleichzeitige Berechnung von Unterbäumen bei der Kraftberechnung (feingranular) und durch gleichzeitige Berechnung mehrerer Körper (grobgranular).

Irregularität: Irregulär, da die Körper ungleichmäßig verteilt sind und der resultierende Baum irregulär ist.

Der Wurzelknoten ist typischerweise 1 700-unbalanciert, also sehr ungleichmäßig. Der Gesamtbaum ist (0,64; 1,00) / (0,65; 1,50) / (0,87; 3,0)-balanciert. Um mindestens 66% der Knoten abzudecken, ist eine Unbalanciertheit von 1,95 nötig.

Rekursion: Zwei maximal 8-fach rekursive Prozeduren (Baumdurchlauf für alle Körper und Kraftberechnung für Einzelkörper).

Rekursionsbaum: Abbildung 6.2 zeigt einen beispielhaften Rekursionsbaum.

Größe: ca. 2 450 LOC

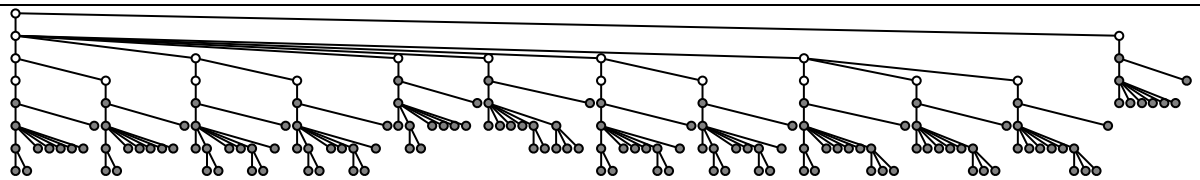


Abbildung 6.2: Barnes Hut — Rekursionsbaum für 12 Körper. Die beiden Rekursionshierarchien sind durch verschiedene Grauwerte der Knoten gekennzeichnet.

Quelle: Abgeleitet aus einem Benchmark von Olden [16], basierend auf dem Originalcode von J. Barnes [5].

Eingabedaten: Zufällig nach J. Barnes generierte Galaxien mit n Körpern, fester Startwert des Zufallsgenerators zur Gewährleistung der Vergleichbarkeit.

Besonderheiten: Es gibt mehrere Iterationen der Berechnung (Zeitschritte). Der Algorithmus ist auf zwei Ebenen rekursiv, was bei Parallelisierung beider Ebenen zu feinkörniger Parallelität führt, während die Parallelisierung nur der obersten Ebene eine gröbere Granularität bewirkt. Das macht den Benchmark zu einem guten Testfall für die automatische Entscheidung, welche Rekursionsebenen parallelisiert werden sollen.

[Die Berechnung der Kräfte für alle Körper erfolgte im Original-Code in einer Schleife, da die Körper sowohl in den Baum einsortiert als auch in einem Feld gehalten sind. Für die Versuche wurde die Schleife durch ein rekursives Durchlaufen des Baums ersetzt — die Begründung ist, daß der Baum im Programm bereits vorhanden ist und ein Programmierer mit dem Wissen, daß das System Rekursionen parallelisiert, genau dieses Durchlaufen des Baums statt der Schleife wählen würde.]

6.1.2 Eigenvalue

Anwendung: Mathematik — Eigenwerte von symmetrischen tridiagonalen Matrizen werden durch rekursive Intervallteilung berechnet.

Algorithmus: Durch ein mathematisches Verfahren läßt sich für diese Art von Matrizen recht einfach die Zahl der Eigenwerte ermitteln, die in einem gegebenen Intervall liegen.

Daraus leitet sich direkt ein Teile-und-Herrsche Algorithmus ab: Halbiere das Intervall solange, bis sich die Zahl der enthaltenen Eigenwerte ändert, und behandle die sich ergebenden Intervallhälften rekursiv weiter. Abbruchbedingung ist das Unterschreiten einer Mindestgröße des Intervalls, womit der enthaltene Eigenwert bestimmt ist. Aufgrund der Rechengenauigkeit (`double`) kann es vorkommen, daß dicht beieinanderliegende Eigenwerte nicht weiter aufgelöst werden können.

Abbildung 6.3 visualisiert den Ablauf.

Parallelität: Das Programm ist durch jeweils gleichzeitige Ausführung der beiden Rekursionsäste sehr gut parallelisierbar. Es gibt keine Datenabhängigkeiten zwischen den Ästen.

Irregularität: Irregulär, da Eigenwerte je nach Matrix sehr verschieden im Intervall verteilt sein können und die Einengung des Intervalls verschieden lange dauern kann.

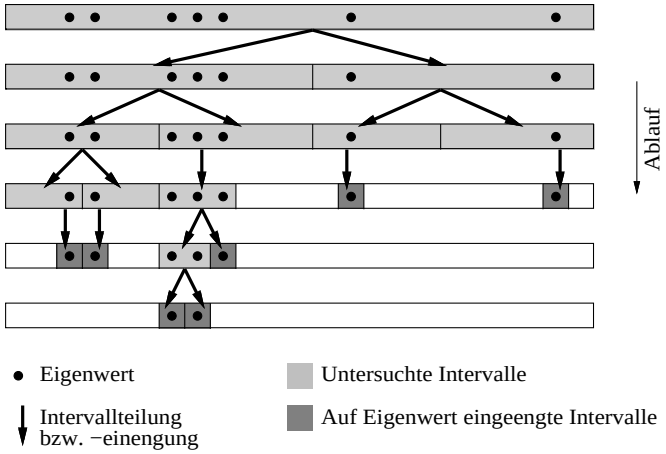


Abbildung 6.3: Eigenvalue — Rekursive Intervallteilung, irreguläre Struktur durch unregelmäßige Verteilung der Eigenwerte.

Der Wurzelknoten ist bei uniformer Eigenwertverteilung typischerweise perfekt 1-balanciert, der Gesamtbaum ist (0,67; 1,00) / (0,82; 1,5) / (1,00; 3,0)-unbalanciert. Eine geometrische Verteilung ergibt eine 130-unbalancierte Wurzel — also extreme Ungleichverteilung auf oberster Ebene — und einen (0,57;1,00) / (0,68;1,5) / (0,95;3,0)-balancierten Baum. Um mindestens 66% der Knoten abzudecken, ist eine Unbalanciertheit von 1,45 nötig.

Rekursion: Eine 2-fach rekursive Prozedur.

Rekursionsbaum: Abbildung 6.4 zeigt zwei beispielhafte Rekursionsbäume für verschiedene Eigenwert-Verteilungen. Man sieht, daß die Eingabedaten einen großen Unterschied der Baumform bewirken — die uniforme Verteilung erzeugt einen perfekten Binärbaum, während eine geometrische Verteilung einen sehr unbalancierten Baum bedingt.

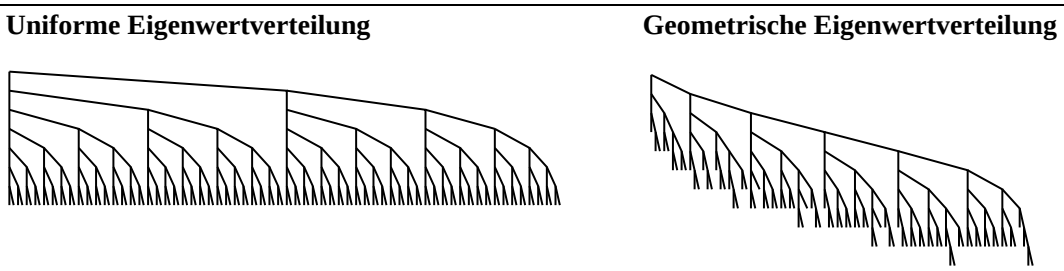


Abbildung 6.4: Eigenvalue — Rekursionsbäume für uniforme und geometrische Eigenwertverteilung bei 128 Eigenwerten, Auflösung der geometrischen Verteilung begrenzt durch Rechengenauigkeit.

Größe: ca. 550 LOC

Quelle: Abgeleitet aus einem Benchmark von [18].

Eingabedaten: Künstliche Grenzfälle der Eigenwertverteilung nach S. Chakrabarti für $n \times n$ Matrizen: Uniforme Gleichvereitung der Eigenwerte auf dem Zahlenstrahl und geometrische konzentrierte Verteilung. Eine Zufallsverteilung kommt der uniformen Verteilung sehr nahe.

Besonderheiten: Eigenvalue ist ein relativ grobgranularer Benchmark, der wie bereits dargestellt je nach Eingabeverteilung ein breites Spektrum von Balanciertheit seiner Rekursionsbäume aufspannt.

Bei der Parallelisierung wurde hier aufgrund des festen Verzweigungsgrades von Zwei immer nur für den linken Teilbaum ein neuer Thread verwendet, während der ursprüngliche Thread mit der Berechnung des rechten Teilbaums fortfährt (Nothread-Annotation, siehe Abschnitt 5.3).

6.1.3 Fractal

Anwendung: Computergrafik — Berechnung der bildlichen Darstellung eines Ausschnitts der Mandelbrot-Menge [66] mittels rekursiver Heuristik.

Algorithmus: Anstatt jeden einzelnen Bildpunkt zu errechnen, kann folgende Heuristik verwendet werden: Wenn der Rand eines Rechtecks im Bild die selbe Farbe hat, wird das ganze Rechteck mit dieser Farbe ausgefüllt. Damit ergibt sich ein rekursives Verfahren, das die Bildfläche solange in Quadranten unterteilt, bis diese ausgefüllt werden können oder nur noch einen Bildpunkt enthalten.

Dieses Vorgehen senkt die Zahl der berechneten Bildpunkte des Ur-Apfelmännchenausschnitts (-2,25; -1,5 ... 0,75; 1,5) bei einer Auflösung von 1 024×1 024 von 1 048 576 auf 278 308. Dieser Effekt verstärkt sich bei größeren Auflösungen. Da es vorkommen kann, daß feine Linien in ein Bild-Rechteck vordringen, aber kein Bildpunkt auf dem Rand des Rechtecks einen Punkt dieser Linie erfaßt, ist das Verfahren nicht genau. Bei erwähnter Auflösung wird aber lediglich ein einziger Bildpunkt im Vergleich zum exakten Verfahren anders gefärbt.

Ein Beispiel des Ablaufs einer rekursiven Unterteilung eines kleinen Bildausschnitts zeigt Abbildung 6.5.

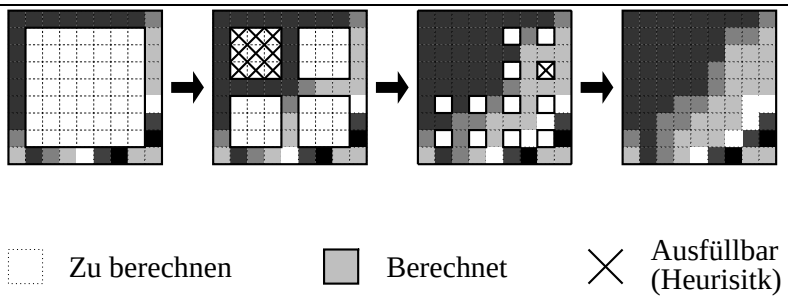


Abbildung 6.5: Fractal — Ablauf der rekursiven Unterteilung und Berechnung eines Bildausschnitts

Parallelität: Die Parallelisierung geschieht durch gleichzeitige Berechnung der Quadranten bei der Rekursion. Es gibt keine Schreibkonflikte.

Irregularität: Abhängig vom Bildausschnitt werden sehr verschiedene Iterationstiefen bei der Bildpunktberechnung an unterschiedlichen Stellen des Bilds erreicht. Außerdem sind die Rechtecke, an denen die Heuristik aktiv wird, nicht vorhersagbar und über das ganze Bild verteilt, so da's der Rekursionsbaum irregulär wird.

Der Wurzelknoten ist typischerweise moderat 1,5-unbalanciert, der Gesamtbaum ist (0,73; 1,00) / (0,74; 1,5) / (0,76; 3,0)-unbalanciert, also bis in unterste Ebenen unbalanciert. Um mindestens 66% der Knoten abzudecken, reicht eine Unbalanciertheit von 1,0 aus.

Rekursion: Eine 4-fach rekursive Prozedur für die Berechnung eines Quadranten.

Rekursionsbaum: Abbildung 6.6 zeigt einen beispielhaften Rekursionsbaum.

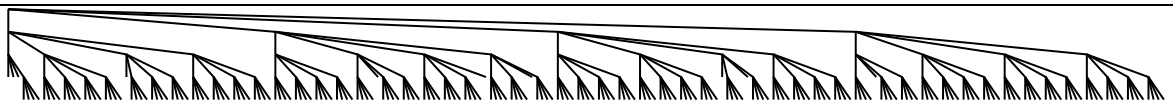


Abbildung 6.6: Fractal — Rekursionsbaum für den Ur-Bildausschnitt der Mandelbrotmenge bei Auflösung 32×32.

Größe: ca. 650 LOC

Quelle: Eigener Code, basierend auf einer Idee aus einem Fractal-Programm für den Atari ST von K. Schneider.

Eingabedaten: Fünf verschiedene Ausschnitte des Apfelmännchens mit $n \times n$ Pixeln, siehe Anhang A.4.

Besonderheiten: Entgegen dem ersten Eindruck handelt es sich um kein Spielzeugbeispiel, da für Fraktale auch in der Realität viele CPU-Stunden aufgewandt werden. Die rekursive Heuristik spart auch bereits sequentiell Rechenzeit und ermöglicht eine elegante Parallelisierung.

Dasselbe Prinzip ist auch für andere Gebiete anwendbar, in denen ein Ausschnitt abhängig von einer Bedingung oder Heuristik dynamisch verfeinert wird.

[Bei der Parallelisierung der Berechnung der Quadranten werden durch Verwendung der Nothread-Annotation nur drei Threads erzeugt, der vierte rechnet den verbleibenden Quadranten aus.]

Dieser Benchmark zeigt auf anschauliche Weise die Korrespondenz zwischen Arbeit (d.h. Bildausschnitt) und Rekursionsbaum, wie der Anhang grafisch ausführt.

6.1.4 Power

Anwendung: Simulation — Strompreise werden in einem baumförmigen Verteilernetz durch Rekursion entlang des Baums berechnet. Das Kraftwerk befindet sich an der Wurzel und die Kunden an Blättern.

Algorithmus: Die Preisberechnung erfolgt durch Propagieren einer Kosteninformation von der Wurzel zu den Blättern, lokale Optimierungen in den Blättern und Zurückpropagieren der entstehenden Nachfrage zur Wurzel. Diese Berechnung wird wiederholt, bis eine Konvergenz eintritt. Es werden verschiedene Faktoren wie z.B. die Stromleitungsverluste auf den verschiedenen Ebenen des Baums berücksichtigt.

Die zentrale Datenstruktur ist ein Baum mit verschiedenen Ebenen von Knotentypen: Ausgehend von der Wurzel (*root*), die das Kraftwerk darstellt, folgt die Ebene der Zuleiter (*feeders*), die sich in Seitenäste (*laterals*) aufspalten, an denen weitere Verzweigungen (*branches*) erfolgen, welche schließlich zu den Blättern (*leafs*) des Baums führen, den simulierten Endkunden.

Den Knotentypen entspricht jeweils eine rekursive Prozedur, die diese Art von Knoten behandelt und Aufrufe an Knoten (Prozeduren) tiefer in der Hierarchie enthält. Dabei fließt auch Information des Nachbarknotens in der Hierarchie ein, wie der nächste Absatz erläutert und Abbildung 6.7 verbildlicht.

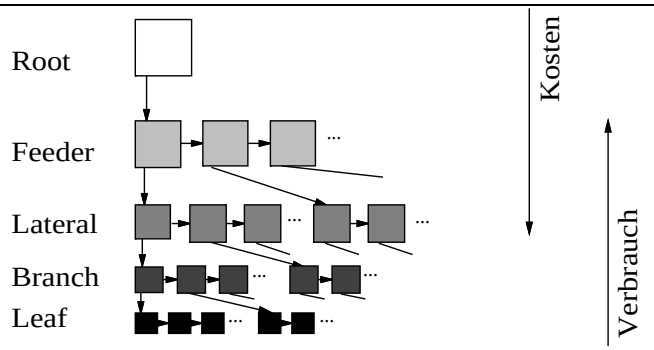


Abbildung 6.7: Power — Hierarchie der Knotentypen in den Baumebenen, Propagation von Preis und Verbrauch. Die Pfeile stellen Verzeigerungen der Datenstruktur dar, die gleichzeitig den algorithmischen Abhängigkeiten entsprechen.

Parallelität: Parallelisierung durch Überlappung der Berechnungen für einen Knoten mit denen der Unterbäume (Art: $feeder(x) = feeder(x-1) + my\_lateral()$, wobei $feeder(x-1)$ und $my\_lateral()$ parallel laufen können) — siehe Algorithmus.

Irregularität: Irregulär da Baum irregulär (s.u.).

Der Wurzelknoten ist typischerweise 7,3-unbalanciert, also unausgewogen. Der Gesamtbaum ist (0,80; 1,00) / (0,81; 1,5) / (0,83; 3,0)-unbalanciert. Um mindestens 66% der Knoten abzudecken, reicht eine Unbalanciertheit von 1,0 aus.

Rekursion: Drei rekursive Prozeduren für die drei Typen von Knoten, hierarchisch (zuerst Rekursion 1, dann 2, dann 3, dann Blätter). Jeweils nur ein rekursiver Aufruf an die aktuelle Prozedur selbst, da Nachfolgeknoten pro Baumebene in Listen vorliegen, und ein Aufruf an die Prozedur der nächsten Hierarchiestufe.

Rekursionsbaum: Abbildung 6.8 zeigt einen beispielhaften Rekursionsbaum.

Größe: ca. 1 200 LOC

Quelle: Abgeleitet aus einem Benchmark von Olden [16], seinerseits abgeleitet aus [65].

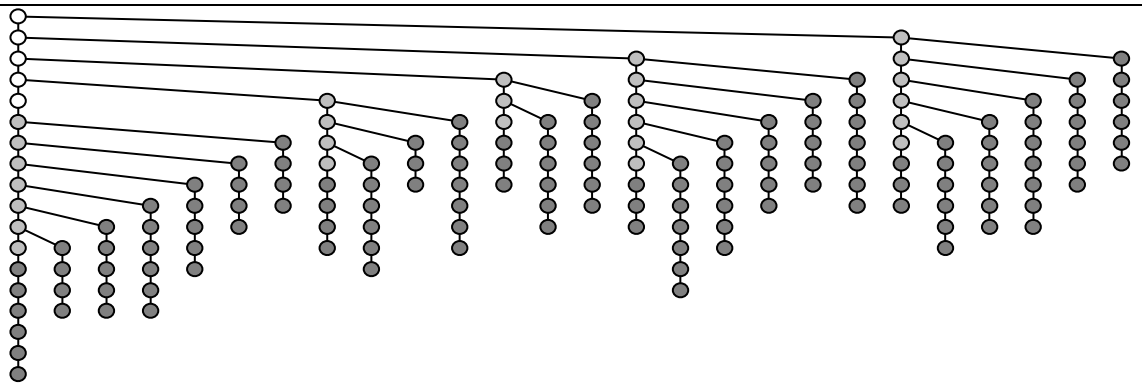


Abbildung 6.8: Power — Rekursionsbaum für Eingabegröße 5,5,5,5. Die verschiedenen rekursiven Prozeduren sind durch verschiedene Graustufen der Knoten gekennzeichnet, die Ebene der Blätter ist nicht rekursiv und daher nicht abgebildet.

Eingabedaten: Verschieden große Bäume mit zufällig variiertem Verzweigungsgrad, fester Startwert des Zufallsgenerators zur Vergleichbarkeit.

Besonderheiten: Es gibt mehrere Iterationen der Berechnung, bis die Konvergenz eintritt. Der Algorithmus weist drei Hierarchiestufen in der Rekursion auf, die jeweils eigene Funktionen beinhalten. Er ist daher ein guter Testfall für die automatische Auswahl von zu parallelisierenden Hierarchieebenen.

[Im ursprünglichen Quellcode waren einige Prozeduren rekursiv geschrieben, andere nicht. Für die Versuche wurden alle Prozeduren, die auf dem Baum arbeiten, konsequent rekursiv formuliert. Außerdem war der Beispiel-Baum völlig regulär, was kaum einem realen Stromnetz entspricht. Daher wurden zufällige Schwankungen von $\pm 50\%$ im Verzweigungsgrad eingebaut, was in irregulären, realistischeren Bäumen resultiert. Für die Messungen wurde der Zufallsgenerator mit einem festen Startwert eingesetzt.]

6.1.5 Queens

Anwendung: Kombinatorik — Alle Lösungen des n-Damen-Problems[1] auf einem $n \times n$ Schachbrett werden berechnet.

Algorithmus: Gradliniges Vorgehen ohne Optimierungen: Rekursiv wird die 1. bis n. Spalte des Schachbretts untersucht. Dabei wird für jede mögliche Position einer Dame in dieser Spalte geprüft, ob sie mit der Plazierung von Damen in den bisherigen Spalten kollidiert. Ist dies nicht der Fall, so werden für diese Plazierung rekursiv die nächsten Spalten überprüft. Alle dermaßen erhaltenen Lösungen werden in einer Baumstruktur für die spätere Ausgabe aufgezeichnet.

Abbildung 6.9 zeigt diesen Ablauf exemplarisch für einen Teilbaum der Berechnung eines 4×4 Schachbretts.

Parallelität: Parallelisierung durch gleichzeitiges rekursives Prüfen aller n Möglichkeiten, die jeweils nächste Dame zu setzen.

[1] Plazierung aller n Damen so, daß sie sich nicht gegenseitig schlagen können, also paarweise weder in der selben Spalte noch Zeile noch Diagonale stehen.

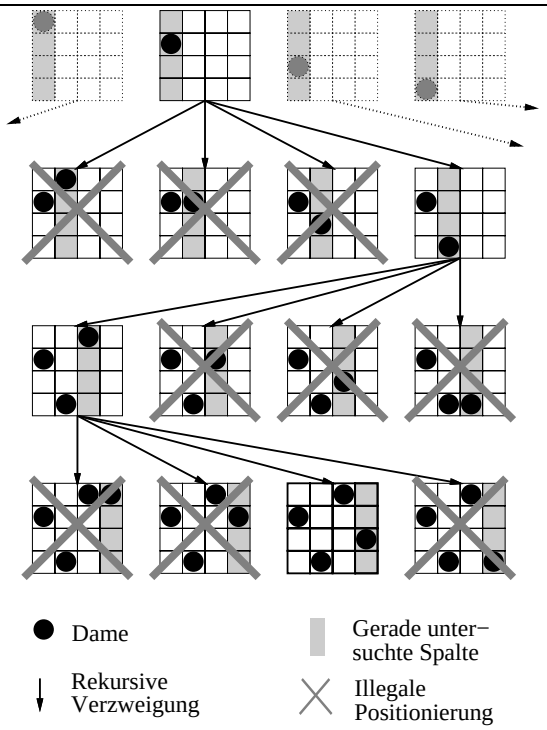


Abbildung 6.9: Queens — Ablauf der rekursiven Berechnung für ein 4×4 Schachbrett (restliche oberste Verzweigungen der Übersichtlichkeit wegen ausgeblendet)

Irregularität: Irregulär, da der entstehende Rekursionsbaum aufgrund der Plazierungsregeln irregulär ist.

Der Wurzelknoten ist typischerweise 1,15-unbalanciert, der Gesamtbaum ist (0,38; 1,00) / (0,38; 1,5) / (0,38; 3,0)-unbalanciert und damit über weite Bereiche hinweg unbalanciert, was dem „ausgefransten“ Aussehen des Rekursionsbaums Rechnung trägt. Um mindestens 66% der Knoten abzudecken, muß eine Unbalanciertheit von 15 in Kauf genommen werden!

Rekursion: Eine n-fach rekursive Prozedur.

Rekursionsbaum: Abbildung 6.10 zeigt einen beispielhaften Rekursionsbaum.

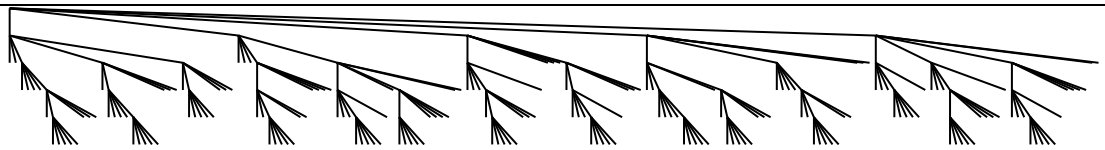


Abbildung 6.10: Queens — Rekursionsbaum für $n = 5$.

Größe: ca. 450 LOC

Quelle: Abgeleitet aus einem Benchmark von Cilk [91].

Eingabedaten: Implizites n-Damenproblem für ein $n \times n$ Schachbrett.

Besonderheiten: Am feinsten granularer Benchmark der Suite, verdeutlicht den Mehraufwand bei der Threaderzeugung.

[Die ursprüngliche Form des Benchmarks berechnete nur eine einzige Lösung des Problems und brach dann ab, was im Parallelen einen indeterministischen Ablauf ergab. Für REAPAR wurde das Programm so umgeschrieben, daß deterministisch alle Lösungen ermittelt und aufgezeichnet werden.]

6.1.6 Beschreibung der Kenngrößen

Folgende Kenngrößen wurden herangezogen, um einen Benchmark und sein Ablaufverhalten zu charakterisieren — der Begriff „Kenngröße“ ist hier global gemeint und entspricht nicht wörtlich den in Kapitel 4 eingeführten Größen:

Benchmark: Name des Benchmarks

Eingabegröße: Größe der Eingabedaten in Einheiten des Programms, z.B. Dimension der Matrix bei Eigenvalue, Verzweigungsgrade verschiedener Hierarchiestufen bei Power oder Zahl der Körper bei Barnes Hut.

Eingabeart: Weitere Differenzierung der Eingabe, z.B. Art der Eigenwerte-Verteilung (uniform, geometrisch) bei Eigenvalues, Bildausschnitt bei Fractal.

Sequentielle Laufzeit: Laufzeit des instrumentierten Programms ohne jegliche Thread-Konstrukte (Strategie Neverever).

Blätter: Gesamtzahl der Blätter in allen Rekursionsbäumen, direkt aus dem Laufzeitprofil abgelesen (Verzweigungsgrad 0). Wenn es mehrere Rekursionshierarchien für die Parallelisierung gibt, sind die Blattzahlen aller Hierarchien aufgeführt.

Knoten: Zahl der Knoten im Rekursionsbaum (ohne Blätter), ditto mit Grad > 0.

Millisekunden pro Blatt: Ungefähre Dauer der Berechnung für ein Blatt (*Laufzeit/Blattzahl*) als Maß für die Granularität des Benchmarks, wiederum nach eventuellen Hierarchien unterschieden.

Rekursionstiefe: Tiefe des Rekursionsbaums bzw. einzelner Hierarchiestufen gemessen ab dem Wurzelknoten (Tiefe 0), jeweils maximale und am häufigsten vertretene.

Verzweigungsgrad: Entsprechende Werte für den Verzweigungsgrad.

Speicher: Maximaler Speicherverbrauch des Problems in Megabytes (1 048 576 bytes).

6.1.7 Tabelle der Kenngrößen

Auf einem einzelnen Prozessor einer Sun SPARCstation 20 mit 100MHz HyperSPARC Prozessor ergaben sich für die sequentiellen Läufe der Benchmarks die in Tabelle 6.1 aufgeführten Werte (Median von je drei Messungen). Der Rechner verfügte über 160MB realen und 202MB virtuellen Speicher, die Prozessoren hatten jeweils 256kB Cache. Als Betriebssystem diente SunOS 5.5.1 (Solaris 2.6) mit den zugehörigen Threadbibliotheken, Übersetzer war `gcc` Version 2.7.2.2.

Bei der Betrachtung der Tabelle fallen einige Punkte auf:

Tabelle 6.1: Kenngrößen der Benchmarks für verschiedene Eingaben auf 100MHz HyperSPARC CPU.

Benchmark	Eingabe Größe	Art	Laufzeit seq. (s)	Blätter	Knoten	ms/Blatt	Rek.tiefe max/freq	V.grad max/freq	Speicher (MB)
Barnes	1 024		8	[a] 10 234	5 119	0,781	10/6	8/0	1,00
Hut				638 993	185 162	0,012	8/4	8/0	
	16 384		260	163 799	79 074	1,587	12/6	8/0	3,60
				18 676 844	4 849 851	0,013	11/4	8/0	
	131 072		2 957	1 310 720	632 250	22,560	15/8	8/0	23,23
				212 560 157	53 457 247	0,013	15/4	8/0	
Eigenvalue	1 500	u[b]	53	1 500	1 499	35,33	11/10	2/0	0,90
	1 500	g	18	[c] 872	871	20,64	50/46	2/0	0,86
	9 000	u	1 833	9 000	8 999	203,66	14/13	2/0	1,59
	9 000	g	118	1 118	1 117	105,55	55/53	2/0	1,00
	36 000	u	37 000	36 000	35 999	1 027,77	16/15	2/0	4,18
	36 000	g	3 800	1 152	1 151	3 298,61	55/51	2/0	1,41
Fractal	512	1[d]	7	18 481	6 160	0,378	8/8	4/0	1,02
	1 024	1	20	51 532	17 177	0,388	9/9	4/0	1,77
	4 096	1	164	410 581	136 860	0,399	11/11	4/0	16,77
	13 000	1	2 828	[e] 3 531 571	1 177 190	0,801	13/13	4/0	>180
Power	7, 15,		3	[f] 41	123	73,17	3/3	2/2	0,89
	3, 7			164	2 419	18,29	21/12	2/2	
				2 583	5 330	1,16	24/12	2/2	
	10, 20,		28	21	231	1 333,33	11/10	2/2	1,49
	5, 10			252	4 326	109,38	30/12	2/2	
				4 578	18 858	6,12	36/18	2/2	
	16, 20,		944	171	2 394	5 520,46	14/13	2/2	2,21
	7, 11			2 565	48 564	368,03	27/9	2/2	
				51 129	307 800	18,46	9/3	2/2	
Queens	10		6	312 612	34 815	0,0256	10/8	10/0	1,00
	11		35	1 639 781	164 246	0,0213	11/9	11/0	2,29
	12		234	9 247 680	841 989	0,0283	12/10	12/0	9,14
	13		1 580	55 140 425	4 601 178	0,0298	13/11	13/0	48,92

[a] Erste Zahl sind die Blätter der 1. (Computesubtree) Hierarchie, zweite die der 2. (Walksub).

[b] u=uniforme Eigenwertverteilung, g=geometrische Verteilung

[c] Double-Genauigkeit begrenzt Blattzahl bei geometrischer Datenverteilung — sollte rein mathematisch auch gleich N sein.

[d] Die Bildausschnitte 1-5 sind in Abschnitt A.4 grafisch dargestellt.

[e] Fast nur Swapping gemessen, mehr Speicher verwendet als physikalisch vorhanden — 1025 User sec aber 2828 Wallclock time!

[f] Verschieden lange Iterationen. Erster Wert sind die Blätter auf der 1. (Feeder) Ebene, zweiter die der 2. (Lateral) Ebene, dritter die Gesamtblätter, jeweils über alle Iterationen gemessen

- Die Laufzeit der Benchmarks deckt ein breites Spektrum ab. Wie sich später zeigt, werden sogar Programme mit Laufzeiten im Sekundenbereich effizient parallelisiert.
- Barnes Hut und Queens haben um zwei Größenordnungen umfangreichere Rekursionsbäume als die anderen Benchmarks, sind also als sehr feingranular zu bezeichnen. Bei Barnes Hut läßt sich durch Ausblenden der unteren Rekursionshierarchie (`walksub`) die Granularität auf normalere Werte erhöhen.
- Die Blattlaufzeiten von Fractal sind ebenfalls konsistent gering ($< 1ms$), und zwar fast unabhängig von der Problemgröße.
- Eigenvalue ist der grobgranularste Benchmark. Nur Power mit Ausblenden aller Rekursionsebenen bis auf die oberste ist vergleichbar grobgranular, mit Blattlaufzeiten im Sekundenbereich.
- Die Rechenzeit pro Blatt schwankt bis auf Queens und Fractal erheblich mit der Problemgröße.
- Bei Eigenvalue hängt die Rekursionstiefe sehr stark mit der Art des Problems zusammen. Bei geometrischer Eigenwertverteilung wird sie hauptsächlich von der Rechengenauigkeit bei der Einengung des Intervalls begrenzt.

Besonders die Granularität wirkt sich auf den Erfolg der verschiedenen Parallelisierungsstrategien aus, wie die Ergebnisse der nächsten Seiten zeigen.

6.2 Thesen, Experimente und Resultate

Vor diesem Hintergrund greifen die folgenden Abschnitte jeweils eine (Unter-)These der Arbeit auf, beschreiben den Aufbau des Experiments und erläutern die Ergebnisse. Um die Vergleichbarkeit zu garantieren, beziehen sich alle Laufzeiten auf den erwähnten Rechner mit 100MHz HyperSPARC-Prozessoren, auch für sequentielle Vorgänge wie die Instrumentierung und die Strategiewahl.

Es werden die folgenden Fragen untersucht:

- Welche Beschleunigungen werden durch die Parallelisierungsstrategien erreicht?
- Wie sind diese Ergebnisse im Vergleich mit verwandten Systemen einzuordnen?
- Was ist die Qualität der Strategiewahl durch Tiefenheuristik bzw. durch Simulation, wie schneiden die gewählten Strategien in der Realität ab?
- Gibt es eine universelle Strategie, die für alle Probleme gute Ergebnisse erzielt?
- Wie abhängig von den Eingabedaten ist eine gewählte Strategie, d.h. bringt eine Strategie auch für andere Eingaben eine gute Leistung?
- Wie wirkt sich die Parallelisierung tieferer Hierarchieebenen im Vergleich zu ihrer sequentiellen Ausführung aus?
- Welchen Einfluß auf die Leistung hat die Nothread-Annotation?
- Wie hoch ist der Laufzeitzuwachs durch die Datensammlung?
- Wie lange dauert die Quellcodetransformation im Vergleich zum eigentlichen Programmablauf?
- Ist die Laufzeit der Strategiewahl im Vergleich zur Programmlaufzeit klein genug?
- Welche Ergebnisse bringt die Kombination von Strategien?

Außerdem stellen weitere Abschnitte die Ergebnisse von Benchmarkläufen auf Maschinen mit bis zu 30 Prozessoren dar und präsentieren eine Fallstudie zum Vergleich von REAPAR mit einer Handparallelisierung.

6.2.1 Effiziente Beschleunigung irregulärer rekursiver Programme

Die Grundthese dieser Arbeit ist, daß sich irreguläre rekursive Programme mit geeigneten Parallelisierungsstrategien effizient auf Parallelrechnern ausführen lassen, d.h. eine gute Beschleunigung erzielen.

6.2.1.1 Versuchsaufbau und Messungen

Um dies zu belegen und gleichzeitig den Raum der Möglichkeiten für die automatische Strategiewahl abzustecken, wurden die Laufzeiten der Benchmarks für alle möglicherweise sinnvollen Strategien gemessen. Dazu wurden eventuell nötige Annotationen (Needresults, Nothread oder Noparallel) in die Programme eingefügt und der Quellcode durch das System automatisch parallelisiert. Die resultierenden threadparallelen Programme wurden für die Strategien Neverever, Never, Always, First 1–7, Keep 1–20, Active 1–20 und Depth 1–20 mit verschiedenen Eingabedaten in jeweils drei Läufen gemessen.

Im Detail gemessen wurden:

- Laufzeit des Gesamtprogramms in Sekunden (*wallclock time*) zur Feststellung der Beschleunigung — Basis ist die Laufzeit mit der Strategie Neverever. Eine Laufzeit von Null steht dabei für einen abgebrochenen Programmlauf, z.B. weil die Zahl der erzeugten Threads die Betriebssystemgrenzen überstieg.
- Summe der auf allen Prozessoren verbrauchten Rechenzeit in Sekunden (*user time*) und
- verbrauchte Systemzeit auf allen Prozessoren in Sekunden (*system time*) als Maß für den Mehraufwand der Parallelisierung mit kritischen Abschnitten, Sperren und Threadbehandlung.
- Zahl der während des Programmlaufs erzeugten Threads (wird für abgebrochene Läufe *nicht* auf Null gesetzt, um die Entwicklung beurteilen zu können).
- Auslastung der Maschine in Prozent (als Summe der einzelnen Prozessorauslastungen).

Yusätzlich wurde das Laufzeitprofil aufgezeichnet, da es praktisch keine Auswirkungen auf die Beschleunigung hat und weitergehende Daten liefert.

6.2.1.2 Erklärung der Meßgraphen

Im folgenden werden häufig die Ergebnisse von Laufzeitmessungen als Grafiken dargestellt. Diese Graphen zeigen auf der x-Achse die verschiedenen Strategien, ggf. mit Parametern, und auf der y-Achse die gemessene Größe. Da aufgrund der Datenmenge die Beschriftung nicht immer angemessen groß sein kann, aber der Aufbau der Graphen immer gleich ist und sich leicht ein visueller Gesamteindruck ergibt, zeigt Abbildung 6.11 einen typischen Graphen mit Erläuterung seiner Elemente.

Die anderen Graph-Typen sehen entsprechend aus. Die y-Achsen sind bei der Beschleunigung und Last fest, um eine sofortige Einschätzung der Leistung zu geben ($y \in 0 \ldots n+1$

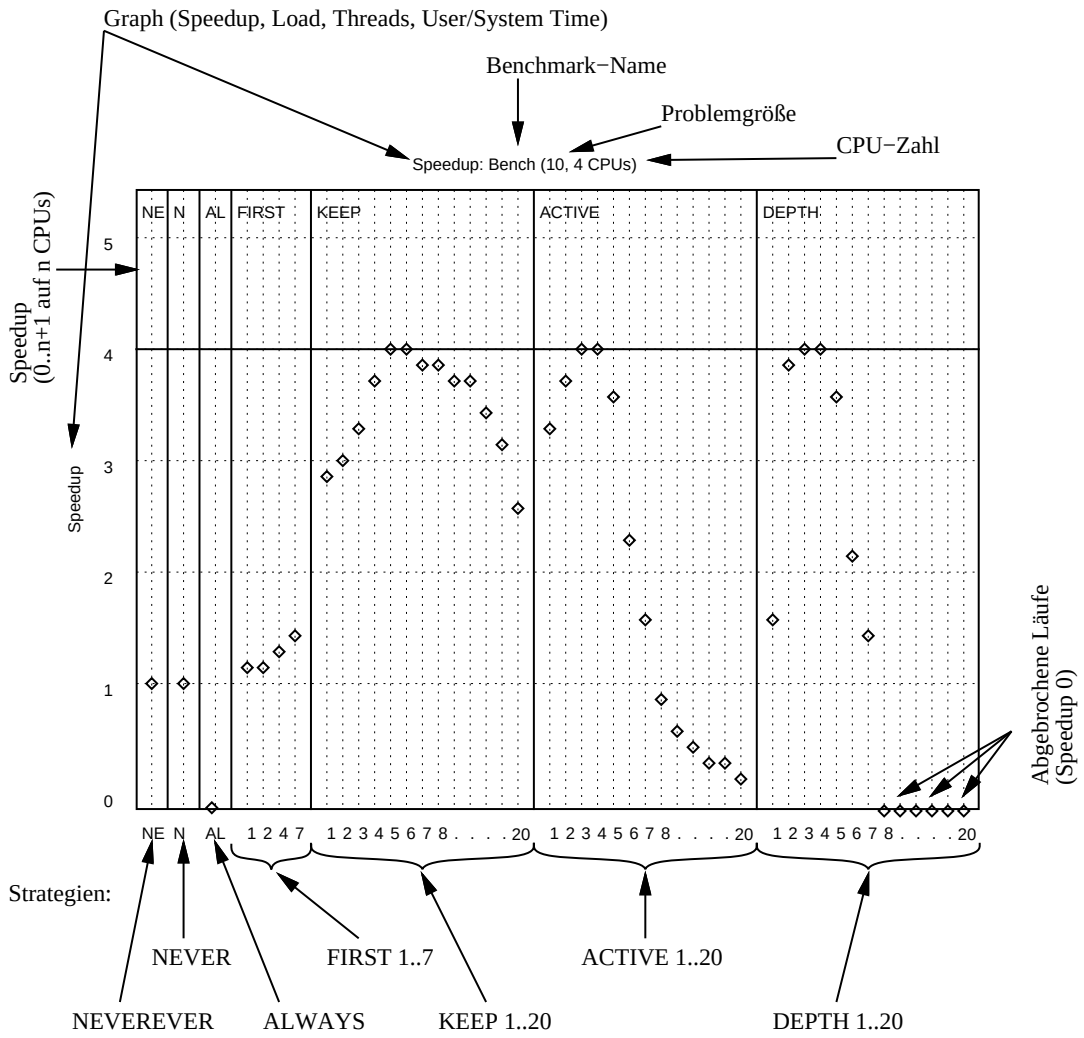


Abbildung 6.11: Elemente eines Meßgraphen.

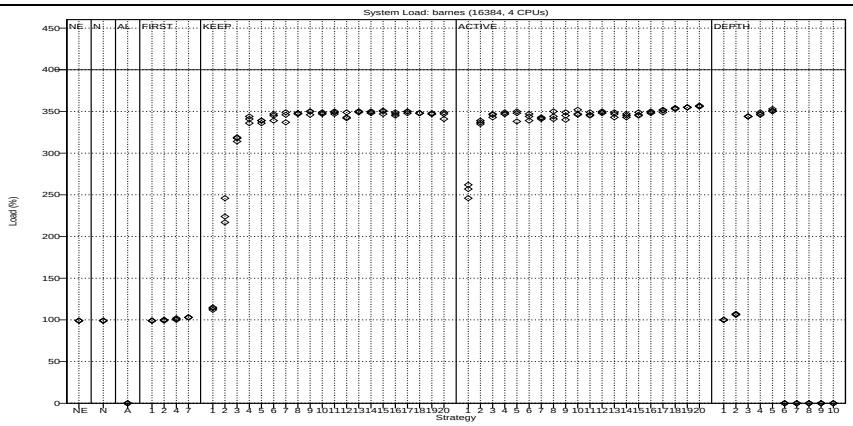
bei n Prozessoren bei der Beschleunigung bzw. 0 . . .n00% bei der Last), hängen bei Threadzahl und Zeitmessung aber von den beobachteten Meßwerten ab. Der Zeitmessungs-Graph zeigt sowohl die produktive Benutzerzeit als Rauten als auch die Mehraufwand bedeutende Systemzeit als Kreuze. Bei Beschleunigung und Last sind horizontale Linien bei n bzw. $n00\%$ durchgezogen, um die Orientierung auch bei kleiner Schriftgröße zu erleichtern.

6.2.1.3 Diskussion einer Meßreihe

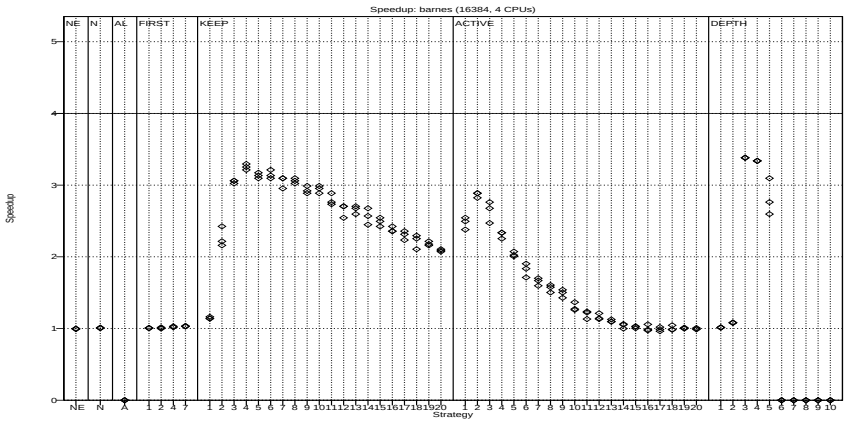
Die Graphen der Meßgrößen geben Einblicke in die Zusammenhänge zwischen Problem, Strategie und erreichter Beschleunigung. Anhand von Abbildung 6.12 diskutieren wir beispielhaft die Ergebnisse der Meßreihe für den Barnes Hut Benchmark mit Eingabegröße 16 384 auf vier Prozessoren:

- Die Auslastung des Rechners erreicht für viele Strategien Werte bis zu 350%, d.h. die Parallelisierung nutzt die Ressourcen aus. Strategien, die zuwenig Parallelismus erzeugen, sind die First-Strategien und Keep sowie Depth mit zu kleinem Strategieparameter. Die Active-Strategie erzeugt bereits mit Parameter 1 eine hohe Zahl paralleler Aktivitäten. Never und Neverever benutzen definitionsgemäß nur einen Prozessor.

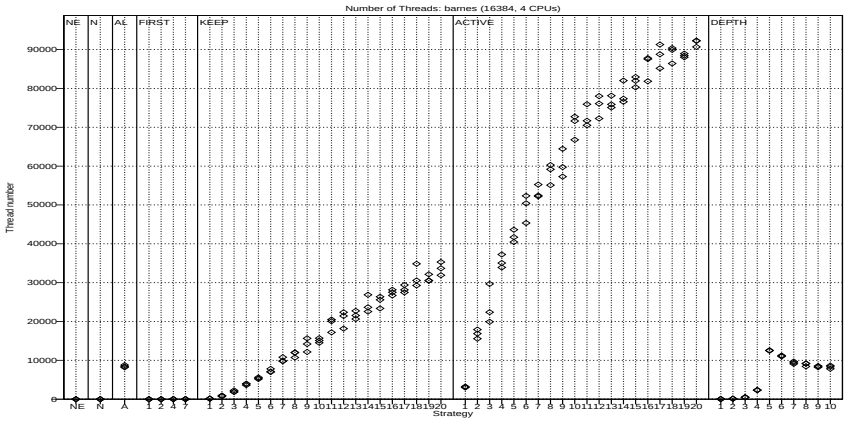
Last:



Beschleunigung:



Threadzahlen:



Zeiten:

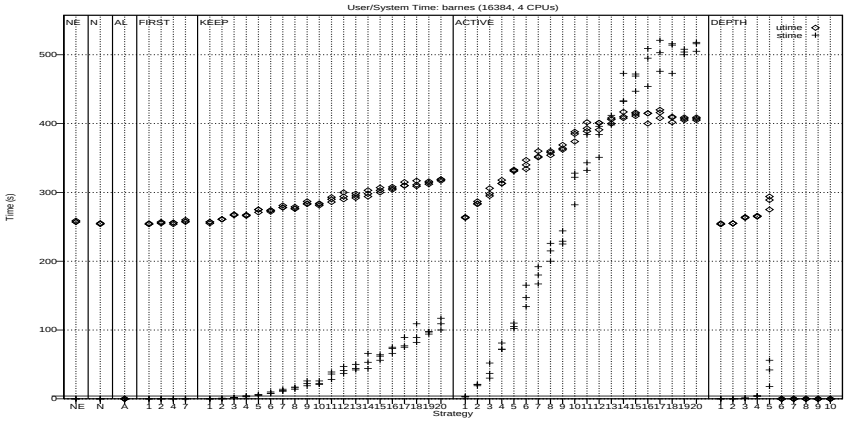


Abbildung 6.12: Meßgraphen für Barnes Hut, Eingabegröße 16 384, Werte von jeweils drei Messungen. Die Zusammenhänge werden im Text diskutiert, wichtig ist hier der grafische Eindruck der übereinanderstehenden Kurven und nicht die Lesbarkeit der Einzelwerte.

- Einige Strategien zeigen eine Last und Beschleunigung von Null. Dies tritt auf, wenn der Programmlauf aufgrund eines Fehlers abgebrochen wurde — wie erwähnt kann Solaris nur etwa 3 000 auf ein `join` wartende Threads verwalten und bricht bei höheren Zahlen ab, was bei Always und bei der Tiefenstrategie mit zu hohen Parametern der Fall ist. Keep und Active erzeugen viele Threads, aber sammeln sie auch schnell genug wieder ein.

- Ein Blick auf die Beschleunigungen erinnert daran, daß eine gute Auslastung nicht automatisch eine gute Beschleunigung bedeutet. Die Tiefenstrategie mit Parameter 3 und 4 schneidet geringfügig besser ab als Keep 4 und Keep 5, wobei der Parameterbereich für gute Beschleunigungen bei Depth wesentlich enger ist als bei Keep. Active-Strategien erreichen hingegen nicht einmal eine Beschleunigung von 3 und fallen schnell auf Werte um 1 herum ab.

- Erklärungen für dieses Verhalten gibt der Graph der erreichten Threadzahlen und der eng damit zusammenhängende Graph der verbrauchten Benutzerzeit (Rauten) und Systemzeit (Kreuze):

  - Ein Grund für schlechte Beschleunigung ist eine zu geringe Zahl von Threads, die die Maschine nicht auslasten. Beispiele sind First und Depth 1-2 mit erwähnt kleiner Last.
  - Die Beschleunigung leidet aber auch unter zu hohen Threadzahlen, wie besonders Active deutlich zeigt. Es werden um eine Größenordnung mehr Threads erzeugt als bei den anderen Strategien, was zu einem übermäßigen Ansteigen der Threadverwaltung führt. Dies spiegelt sich in der hohen Systemzeit wieder, die die Benutzerzeit weit übersteigt.
  - Das Ansteigen der Benutzerzeit selbst erklärt sich mit dem Rechenaufwand für die Thread-Hüllprozeduren und ihre Aufrufe, die bei Verwendung von weniger Threads kaum zum Tragen kommen. Bei Barnes Hut als feingranularem Benchmark ist dieses Verhalten besonders deutlich.

- Das schlechte Abschneiden von Depth 2, die bei einem Verzweigungsgrad von bis zu 8 eigentlich genügend Parallelität liefern sollte, läßt sich durch Betrachten des Rekursionsbaums verstehen (Abbildung 6.2): Auf oberster Ebene gibt es nur zwei Äste, von denen einer nur einen sehr kleinen Unterbaum hat, während alle restlichen Knoten im Unterbaum des anderen Asts liegen. Eine Parallelisierung auf dieser Ebene führt also zu einer extremen Ungleichverteilung der Arbeit mit entsprechenden Auswirkungen auf die Beschleunigung.

Wie man sieht, bietet die Kombination der Meßgraphen eine Erklärung für die beobachteten Effekte, wobei zusätzliches Wissen wie das Aussehen des Rekursionsbaums weitere Einblicke liefert.

Im folgenden werden hauptsächlich die Beschleunigungs-Graphen gezeigt, da sie für den Endbenutzer am relevantesten sind.

6.2.1.4 Erreichte Beschleunigungen

Tabelle 6.2 zeigt für alle Benchmarks die drei besten erreichten Beschleunigungen und die entsprechenden Parallelisierungsstrategien. Wenn mehrere Strategien gleichgut abschnitten, wurden die Strategien mit den kleineren Parametern bevorzugt abgedruckt.

Außerdem ist angegeben, wieviel Prozent der ca. 70 Strategien eine Beschleunigung von mindestens 80% der besten erzielten Beschleunigung erreichen. Ein hoher Wert bedeutet, daß der Benchmark weniger kritisch gegenüber der Strategiewahl ist, ein kleiner Wert läßt der Strategiewahl nur einen geringen Spielraum. Es stellt sich heraus, daß typischerweise 30% der Strategien eine Beschleunigung in diesem Bereich liefern, wobei das grobgranulare Eigenvalue bei uniformen Eingaben besonders unkritisch bezüglich der Strategiewahl ist und der feingranulare Queens-Benchmark nur für sehr wenige Strategien gute Ergebnisse bringt, da hier nur die ersten drei Tiefenstrategien überhaupt gut abschneiden.

Tabelle 6.2: Beste drei Beschleunigungen und entsprechende Strategien für alle Benchmarks auf vier Prozessoren (Beschleunigungen über vier sind aufgrund der Meßgenauigkeit von einer Sekunde bei kurzen Problemen möglich). Die letzte Spalte gibt an, wieviel Prozent der Strategien mindestens 80% der optimalen Beschleunigung erreichen.

Benchmark-	Problem-	1.Platz		2.Platz		3.Platz		> 80%
name	größe	Bes.	Strategie	Bes.	Strategie	Bes.	Strategie	
Barnes Hut	1 024	2,667	Keep 2	2,667	Depth 3	2,667	Keep 3	9%
	16 384	3,382	Depth 3	3,338	Depth 4	3,295	Keep 4	25%
	131 072	3,135	Keep 10	3,122	Keep 8	3,109	Keep 9	39%
Eigenvalue[a]	u-1 500	4,154	Active 1	4,154	Keep 1	4,154	Active 2	94%
	g-1 500	4,500	Active 2	4,500	Active 3	4,500	Active 4	7%
	u-9 000	3,989	Active 2	3,989	Active 3	3,989	Active 4	82%
	g-9 000	4,138	Active 7	4,138	Keep 9	4,138	Keep 12	52%
	u-36 000	4,019	Depth 12	4,017	Keep 2	4,017	Keep 5	85%
	g-36 000	4,027	Active 12	4,022	Keep 10	4,022	Active 13	52%
Fractal[b]	1 1 024-1 024	3,333	Active 1	3,333	Depth 2	3,333	Active 2	37%
	1 4 096-4 096	3,265	Depth 4	3,265	Depth 5	3,200	Active 2	37%
	2 1 024-1 024	3,333	Depth 3	3,333	Keep 3	3,333	Depth 4	19%
	2 4 096-4 096	3,406	Depth 3	3,406	Depth 4	3,406	Depth 5	22%
	3 1 024-1 024	3,500	Depth 2	3,500	Keep 2	3,500	Depth 3	15%
	3 4 096-4 096	2,815	Depth 3	2,815	Depth 4	2,815	Depth 5	24%
	4 1 024-1 024	3,933	Depth 4	3,688	Active 1	3,688	Keep 2	210%
	4 4 096-4 096	3,777	Depth 5	3,759	Depth 6	3,742	Depth 4	48%
	5 1 024-1 024	3,556	Keep 2	3,556	Depth 3	3,556	Keep 3	27%
	5 4 096-4 096	3,664	Depth 4	3,664	Depth 5	3,632	Depth 3	37%
Power[c]	o 10-20-5-10	3,625	Active 3	3,625	Keep 3	3,625	Active 4	73,13%
	o 16-20-7-11	3,739	Keep 10	3,739	Keep 20	3,724	Keep 6	70,15%
	b 10-20-5-10	3,222	Depth 10	2,900	Depth 9	2,900	Depth 11	7,46%
	b 16-20-7-11	3,209	Depth 13	3,167	Depth 12	3,025	Keep 20	52,24%
Queens	10	5,000	Depth 1	5,000	Depth 2	2,500	Depth 3	2,99%
	11	3,889	Depth 1	3,889	Depth 2	3,889	Depth 3	4,48%
	12	3,862	Depth 1	3,862	Depth 2	3,797	Depth 3	5,97%

[a] Jeweils geometrische und uniforme Eigenwertverteilung.

[b] Fünf verschiedene Bildausschnitte, Ausschnitt 1 ist das Ur-Apfelmännchen — im Anhang sind die Ausschnitte genau angegeben und mit ihren Rekursionsbäumen zusammen abgebildet.

[c] „o“ = Ergebnisse mit Parallelisierung nur der obersten Hierarchieebene, „b“ = mit beiden obersten Ebenen — siehe auch Absatz 6.2.6.

6.2.1.5 Vergleich mit der Literatur

Einige der Beschleunigungen lassen sich direkt mit Werten aus der Literatur vergleichen — Tabelle 6.3 zeigt die Ergebnisse von Olden [16] und Cilk [91] gegenüber den Resultaten der vorliegenden Arbeit. Für die Benchmarks Eigenvalue, Fractal und Magic liegen keine Vergleichswerte vor. Mit (*) markierte Einträge sind Bestandteil der Validierungsmenge.

Tabelle 6.3: Vergleich der mit REAPAR erzielten Beschleunigungen mit den veröffentlichten Werten für dieselben Benchmarks unter Olden und Cilk auf 4 Prozessoren.

Benchmark-name	Beschleunigung		
	Olden	Cilk	REAPAR
Barnes Hut	3,00	3,14	**3,38**
Bitonic Sort (*)	2,29	—	**3,36**
Heat (*)	—	3,90	**4,15**
Knapsack (*)	—	**3,60**	3,11
Power	**3,81**	—	3,63
Queens	—	**3,90**	3,89

Da die entsprechenden Benchmarks des REAPAR-Systems aus den Olden-Benchmarks angepaßt wurden, ist ein direkter Vergleich zulässig. Die Messungen von Olden fanden auf einer Connection Machine CM-5 statt und beinhalten die Migration von Prozessen, während REAPAR auf einer SparcStation mit gemeinsamem Speicher ablief. Beim Bitonischen Sortieren gibt Olden nur die Laufzeit des Programmkerns an; die REAPAR Beschleunigung bezieht sich auf das gesamte Programm (für dieses Programm greifen wir ebenso wie für Heat und Knapsack den Ergebnissen der Validierung in Kapitel 7 vor, da diese Benchmarks zur Kontrollmenge des REAPAR Systems gehören).

Die Beschleunigungen von REAPAR liegen deutlich über denen von Olden, bis auf den 5% schlechteren Power Benchmark. Die Verbesserung von 47% beim Bitonic Sort Benchmark kann durch die nötigen Datenbewegungen auf der CM-5 erklärt werden, die bei gemeinsamem Speicher wegfallen.

Cilk berichtet von Messungen von einer Sun Enterprise 5000 SMP Maschine (acht UltraSPARC Prozessoren) mit gemeinsamem Speicher und Threads, also direkt vergleichbar mit der REAPAR Umgebung. Die Benchmark-Suite umfaßt ebenfalls eine Barnes Hut Implementation, die 7% schlechter abschneidet als die Parallelisierung durch REAPAR. Der Queens-Benchmarks von REAPAR ist zwar aus Cilk abgeleitet, berechnet aber alle Lösungen, im Gegensatz zur Cilk-Variante, die nach Auffinden der ersten Lösung abbricht. Ein direkter Vergleich ist also nicht möglich, aber die Beschleunigungen beider Systeme liegen nahe dem Optimum. Für Heat erreicht REAPAR bessere Werte (die überlineare Beschleunigung ist vermutlich auf Cacheeffekte zurückzuführen), lediglich beim extrem feinkörnigen Knapsack-Benchmark schneidet Cilk 15,8% besser ab.

Der Konferenzbeitrag [18], aus dem der Eigenvalue-Benchmark abgeleitet ist, zeigt nur Beschleunigungskurven auf einer Connection Machine CM-5, aber keine exakten Werte. Die Kurven liegen nahe dem linearen Speedup von 4 auf 4 Prozessoren, was REAPAR ebenfalls erreicht.

Zu beachten ist, daß alle erwähnten Systeme im Gegensatz zu REAPAR nicht automatisch parallelisieren, sondern Sprachkonstrukte und zusätzliche Datentypen benötigen, die

der Programmierer explizit setzen muß. Es ist anzunehmen, daß die Programmierer dieser Benchmarks ihr jeweiliges System dabei optimal ausnutzen. Daß REAPAR dennoch so gut abschneidet, unterstreicht nochmals die Qualität der REAPAR-Ergebnisse.

6.2.2 Qualität der Tiefenheuristik

Für die verwendeten 100MHz HyperSPARC Prozessoren wurde die Grenze zwischen grobgranularen und feingranularen Problemen bei einer durchschnittlichen Blattlaufzeit von 1ms gezogen. Dieser Wert wurde, wie in Abschnitt 5.6.1 erläutert, empirisch bestimmt. Grobgranular sind ihm zufolge Barnes Hut (mit Parallelisierung nur der obersten Rekursionshierarchie), Eigenvalue und Power. Als feingranular klassifiziert werden Fractal und Queens.

Für die letzteren beiden wählt die Tiefenheuristik die in Tabelle 6.4 aufgelisteten Strategien, deren reales Abschneiden ebenfalls aus der Tabelle ersichtlich ist. Verwendet wurde dabei die AS-Heuristik. Die LPS-Heuristik liefert besonders für Queens zu pessimistische Tiefenwerte, die in der Realität unnötig viele Threads erzeugen. Wenn mehrere reale Strategien gleich gut abschneiden, ist die entsprechende Tiefenstrategie aufgeführt.

Tabelle 6.4: Durch die AP-Tiefenheuristik gewählte Strategien für die feingranularen Benchmarks und ihr reales Abschneiden verglichen mit der optimalen Strategie (100% = bestmögliche Leistung).

Benchmark	Problemgröße	Beste reale Strategie	Gewählte Strategie	Erreichte Leistung
Fractal	$1\ 1024^2$	Depth 2	Depth 2	100%
	$1\ 4096^2$	Depth 4	Depth 2	92%
	$2\ 1024^2$	Depth 3	Depth 2	86%
	$2\ 4096^2$	Depth 3	Depth 2	80%
	$3\ 1024^2$	Depth 2	Depth 2	100%
	$3\ 4096^2$	Depth 4	Depth 2	87%
	$4\ 1024^2$	Depth 4	Depth 2	83%
	$4\ 4096^2$	Depth 5	Depth 2	87%
	$5\ 1024^2$	Depth 3	Depth 2	90%
	$5\ 4096^2$	Depth 5	Depth 2	84%
Queens	10	Depth 1	Depth 1	100%
	11	Depth 1	Depth 1	100%
	12	Depth 1	Depth 1	100%

Für Fractal mit den verwendeten Problemgrößen wählt die Heuristik immer die Tiefe 2, was für viele Probleme die optimale Leistung erzielt und maximal 20% Verlust gegenüber der perfekten Strategiewahl bringt. Die Strategiewahl für Queens ist durchgehend perfekt.

6.2.3 Realitätsnähe der Simulation

Die als grobgranular klassifizierten Benchmarks sind Barnes Hut, Eigenvalue und Power. Für sie wurden die Rekursionsbäume aufgezeichnet und der Threadablauf wie in Abschnitt 5.6.3 beschrieben simuliert. Die Suche nach guten Strategien in jeder der drei Strategieklassen

wurde jeweils abgebrochen, sobald die Simulation zum erstenmal eine erfolgversprechende Strategie identifiziert hatte.

Die Ergebnisse der Simulation zeigt Tabelle 6.5, in der die für jede Strategieklasse gewählte Strategie, die Reihenfolge der drei Klassen und die Leistung verglichen mit dem realen Abschneiden der gewählten Strategien aufgeführt sind. Die Reihenfolgesortierung der Klassen fand anhand der Zahl der Simulationsschritte statt, d.h. die Strategie mit den wenigsten Schritten wird als global beste (1. Klasse) beurteilt, die mit den zweitwenigsten Schritten als 2. etc. Für manche Probleme gibt es Strategieklassen, bei denen keine Simulation die von den Auswahlheuristiken gestellten Anforderungen erfüllt. Sie sind in der Tabelle entsprechend markiert.

Bei den Simulationen wurden die Gegebenheiten der Benchmarks berücksichtigt, z.B. die Nothread-Annotation bei Eigenvalue, und die Parallelisierung nur der obersten Rekursionshierarchie bei Barnes Hut und Power.

Tabelle 6.5: Durch die Threadablaufsimulation gewählte Strategien für die grobgranularen Benchmarks und ihr Abschneiden verglichen mit der optimalen Strategie (100% = bestmögliche Leistung). Bei mit „—“ gekennzeichneten Einträgen erfüllte keine Strategie dieser Klasse die Auswahlheuristiken.

Benchmark	Problem-größe	Beste reale Strategie	Gewählte Strategie 1. Klasse	2. Klasse	3. Klasse
Barnes Hut	1 024	Keep 2	Keep 5 100%	Depth 3 100%	Active 2 75%
	16 384	Depth 3	Depth 3 100%	Keep 8 100%	Active 1 81%
	131 072	Keep 10	Keep 10 100%	Depth 4 90%	Active 1 81%
Eigenvalue	u-1 500	Active 1	Depth 3 100%	Active 1 100%	Keep 2 100%
	g-1 500	Active 2	Active 2 100%	Keep —	Depth —
	u-9 000	Active 2	Depth 4 99%	Active 2 100%	Keep 2 99%
	g-9 000	Active 7	Active 2 97%	Keep —	Depth —
	u-36 000	Keep 2	Depth 4 97%	Active 2 99%	Keep 2 100%
	g-36 000	Active 12	Active 2 97%	Keep —	Depth —
Power	10 20 5 10	Active 3	Keep 3 100%	Active 3 100%	Depth 9 80%
	16 20 7 11	Keep 10	Keep 2 53%	Active 2 52%	Depth 8 52%

Das perfekte Abschneiden der Strategiewahl bei Barnes Hut und Eigenvalue spricht für die Qualität der Simulation — bei der geometrischen Eigenwertverteilung wird völlig korrekt

die Active-Strategie bevorzugt, während Keep und Depth für alle Simulationen im Parameterbereich 1 . . . 20 von der Auswahlheuristik abgelehnt werden. Einzig bei Power mit hoher Problemgröße verliert die Simulation 47% an Leistung.

Ebenfalls bemerkenswert ist, daß die Zahl der Simulationsschritte, die jeweils für die Reihenfolge der Strategieklassen zugrundegelegt wurde, offenbar ein sehr gutes Kriterium für die Qualität einer Strategie ist, da die erstplazierte Strategie bis auf Barnes Hut 131 072 immer besser abschneidet als die zweite und dritte Klasse.

6.2.4 Nichtexistenz einer universellen Strategie

Die automatische Strategiewahl wäre nicht nötig, wenn es eine einzige Strategie gäbe, die über alle Probleme hinweg gute Beschleunigungen liefert. Dem ist aber nicht so, wie der Vergleich zwischen Fractal, Eigenvalue und Queens zeigt:

Beim sehr feingranularen Queens scheitern alle Strategien, die sich auf die aktuelle Threadzahl beziehen, weil bei ihnen der Mehraufwand für den nötigen kritischen Abschnitt im Verhältnis zur eigentlichen Rechenzeit eines Teilproblems zu groß ist. Bei Fractal gibt es sowohl für die allgemeinen als auch für die Tiefenstrategie Parameter, die zu guten Beschleunigungen führen. Auch bei Eigenvalue mit geometrischer Verteilung ist dies der Fall, aber die optimalen Parameterwerte sind ganz andere als bei Fractal.

Die Graphen in Abbildung 6.13 stellen die entsprechenden Beschleunigungen gegenüber, wobei die Lesbarkeit der genauen Werte weniger wichtig ist als der grafische Gesamteindruck.

6.2.5 Gültigkeit der Strategiewahl für andere Eingaben

Die Strategiewahl bezieht sich immer nur auf das aktuell betrachtete Problem. Damit stellt sich die Frage, wie gut eine Strategie, die für das aktuelle Problem die beste Beschleunigung erzielt, für andere Probleme desselben Benchmarks abschneidet. Ist z.B. die Strategie, die für Eigenvalue mit 1 500 uniform verteilten Eingaben eine optimale Beschleunigung bringt, auch für 36 000 geometrisch verteilte Eingabewerte empfehlenswert?

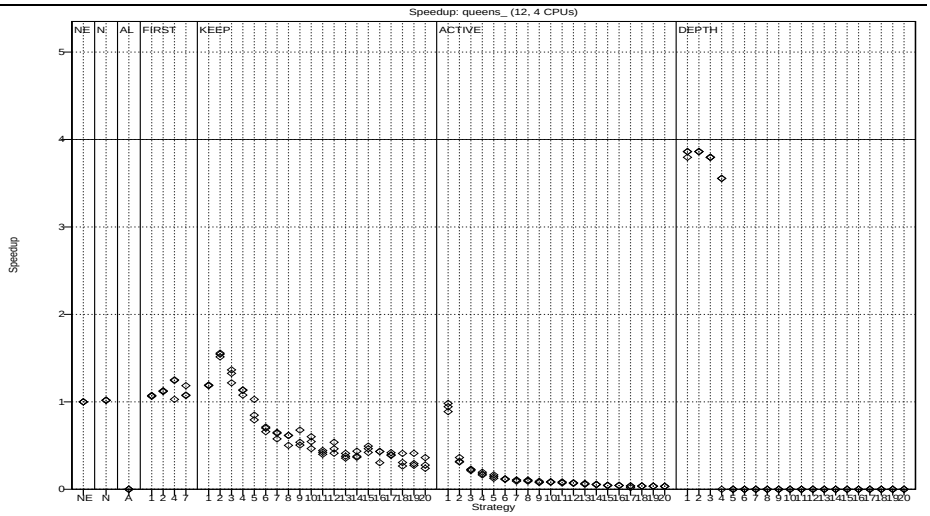
Wenn diese Frage bejaht werden kann, dann ist die Strategiewahl nicht so stark von den genauen Eingabewerten abhängig und kann damit für eine ganze Reihe von Problemen verwendet werden. Eine Änderung in den Eingaben wirkt sich also nicht zu negativ auf die Beschleunigung mit der bereits gewählten Strategie aus.

Um die Frage zu beantworten, wurden die Strategien aller Probleme eines Benchmarks nach absteigender Beschleunigung sortiert und mit Rängen versehen. Gleichgute Strategien erhalten denselben Rang. Außerdem wurde für jede Strategie jedes Problems berechnet, wieviel Prozent der Beschleunigung der optimalen Strategie sie erreicht. Anhand dieser Daten wurde für die jeweils besten drei Strategien eines Problems festgestellt, wie gut diese Strategien für die anderen Probleme des Benchmarks abschneiden.

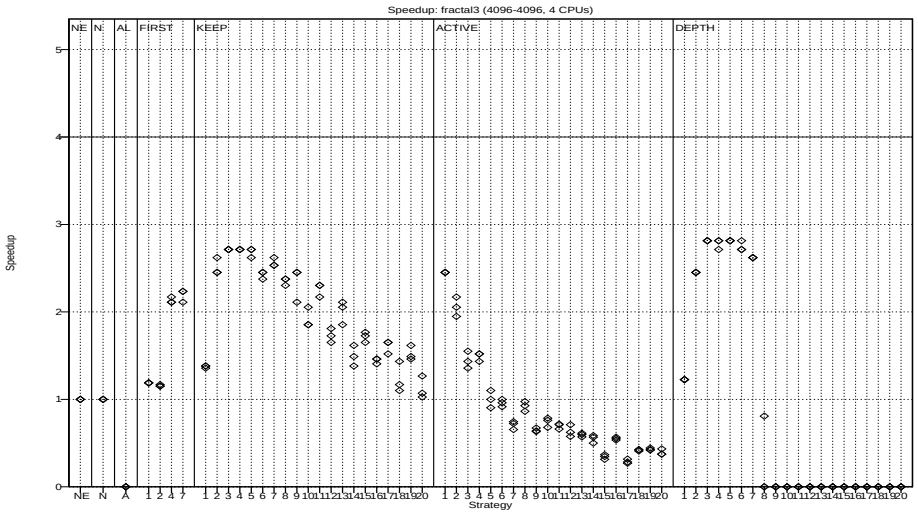
Die Tabellen 6.6 bis 6.10 zeigen das Ergebnis der Vergleiche. In der Zelle (x, y) einer Tabelle steht, wie gut die Strategie des Problems y für das Problem x abschneidet: Ihr Rang unter den Strategien von x wird angegeben, und grafisch wird dargestellt, welchen Prozentsatz der besten Beschleunigung von x die Strategie erreicht. Ein voll ausgefüllter Balken steht dabei für 100%. Besonderheiten der Benchmarks und zusätzliche Erklärungen finden sich in der Überschrift der jeweiligen Tabellen.

Es stellt sich heraus, daß für fast alle Benchmarks die guten Strategien eines Problems auch für die anderen Probleme hohe Leistungen erzielen, typischerweise 80%-100% des Optimums. Damit ist gezeigt, daß eine Strategiewahl auch über das konkrete Problem hinaus

Queens 12:



Fractal 4 096 (Bildausschnitt 3):



Eigenvalue 9 000 geometrisch:

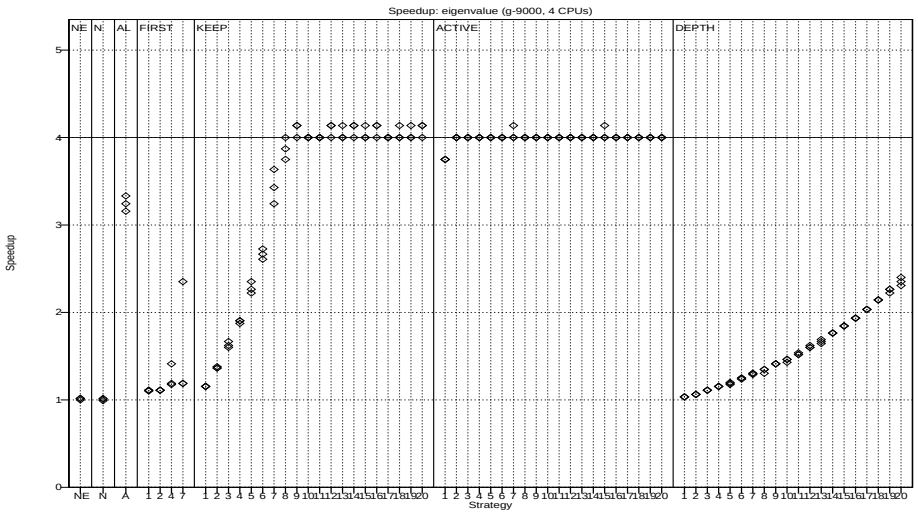


Abbildung 6.13: Gegenüberstellung der Beschleunigungen von drei Benchmarks: Keine Strategie erreicht auch nur annähernd eine gleichermaßen gute Beschleunigung für alle drei Probleme.

Tabelle 6.6: Vergleich der Strategien für Barnes Hut. Die für 1 024 guten Strategien schneiden bei den beiden anderen Eingabegrößen schlechter ab, während die Tiefenstrategien von 16 384 für alle Probleme gute Ergebnisse erzielen.

Problem	Beste Strategien	1 024	16 384	131 072
1 024	Platz 1: keep 2	—	14	29
1 024	Platz 2: active 1	—	13	25
1 024	Platz 3: active 3	—	10	22
16 384	Platz 1: depth 3	1	—	21
16 384	Platz 2: depth 4	1	—	18
16 384	Platz 3: keep 4	1	—	13
131 072	Platz 1: keep 10	3	8	—
131 072	Platz 2: keep 8	3	6	—
131 072	Platz 3: keep 9	3	8	—

Tabelle 6.7: Vergleich der Strategien für Eigenvalue. Die besten Strategien aller Probleme erreichen auch für die anderen Probleme gute oder optimale Beschleunigungen. Dieser grobgranulare Benchmark erzielt für genügend hohe Strategieparameter mit allen Strategien sehr gute Beschleunigungen. Kritisch sind Tiefenstrategien und First, die bei uniformer Eigenwertverteilung sehr gut abschneiden, aber bei geometrischer Verteilung hohe Parameterwerte erfordern, die hier nicht gegeben sind. Dementsprechend schlecht sind ihre Ergebnisse bei den geometrischen Problemen, was aber weniger kritisch ist, weil sie sowieso nie auf dem ersten Platz sind.

Problem	Beste Strategien	g-1 500	g-9 000	g-36 000	u-1 500	u-9 000
g-1 500	Platz 1: active 2	—	2	14	1	1
g-1 500	Platz 2: active 1	—	3	16	1	2
g-1 500	Platz 3: keep 5	—	8	21	1	8
g-9 000	Platz 1: active 7	1	—	4	1	1
g-9 000	Platz 2: active 2	1	—	14	1	1
g-9 000	Platz 3: active 1	2	—	16	1	2
g-36 000	Platz 1: active 12	1	2	—	1	1
g-36 000	Platz 2: keep 10	1	2	—	1	1
g-36 000	Platz 3: active 6	1	2	—	1	1
u-1 500	Platz 1: active 1	2	3	16	—	2
u-1 500	Platz 2: first 1	13	28	43	—	12
u-1 500	Platz 3: first 2	12	28	44	—	13
u-9 000	Platz 1: active 2	1	2	14	1	—
u-9 000	Platz 2: active 1	2	3	16	1	—
u-9 000	Platz 3: depth 5	11	26	39	2	—

Tabelle 6.8: Vergleich der Strategien für Fractal, hier aus Übersichtlichkeitsgründen nur mit den Bildausschnitten 1,3 und 4, jeweils mit 1024×1042 und 4096×4096 Bildpunkten. Das Problem 3 hat den unregelmäßigsten Rekursionsbaum und ist daher hinsichtlich der Strategiewahl am kritischsten, während die besser balancierten Bäume der Probleme 1 und 4 für mehr Strategien sehr gute Beschleunigungen liefern.

Problem	Beste Strategien	1_1024²	1_4096²	3_1024²	3_4096²	4_1024²	4_4096²
1_1024²	Platz 1: active 1	—	5	2	4	2	5
1_1024²	Platz 2: keep 1	—	12	3	19	3	13
1_1024²	Platz 3: first 1	—	16	4	22	5	20
1_4096²	Platz 1: depth 4	1	—	1	1	1	3
1_4096²	Platz 2: active 2	1	—	2	8	3	13
1_4096²	Platz 3: keep 4	1	—	1	2	2	6
3_1024²	Platz 1: depth 2	1	5	—	4	4	17
3_1024²	Platz 2: active 1	1	5	—	4	2	5
3_1024²	Platz 3: keep 1	2	12	—	19	3	13
3_4096²	Platz 1: depth 3	1	2	1	—	2	11
3_4096²	Platz 2: keep 3	1	2	1	—	2	4
3_4096²	Platz 3: keep 2	2	7	1	—	2	6
4_1024²	Platz 1: depth 4	1	1	1	1	—	3
4_1024²	Platz 2: active 1	1	5	2	4	—	5
4_1024²	Platz 3: depth 1	3	14	4	21	—	18
4_4096²	Platz 1: depth 5	1	1	1	1	2	—
4_4096²	Platz 2: depth 6	1	2	2	1	3	—
4_4096²	Platz 3: depth 4	1	1	1	1	1	—

Tabelle 6.9: Vergleich der Strategien für Power. „a“ bezeichnet die Version des Programms mit Parallelisierung nur der oberen Ebene (s.u.), „b“ die auf zwei Ebenen parallele Version. Letztere ist wie in Abschnitt 6.2.6 erläutert wesentlich kritischer in der Hierarchiewahl und liefert daher oft nur 40%-60% des Optimums für die gegebenen Strategien.

Problem	Beste Strategien	a 10-20-5-10	a 16-20-7-11	b 10-20-5-10	b 16-20-7-11
a 10-20-5-10	Platz 1: active 3	—	8	6	26
a 10-20-5-10	Platz 2: depth 10	—	11	1	18
a 10-20-5-10	Platz 3: depth 9	—	12	2	22
a 16-20-7-11	Platz 1: keep 10	1	—	4	8
a 16-20-7-11	Platz 2: keep 6	1	—	5	15
a 16-20-7-11	Platz 3: active 4	1	—	4	19
b 10-20-5-10	Platz 1: depth 10	2	11	—	18
b 10-20-5-10	Platz 2: depth 9	3	12	—	22
b 10-20-5-10	Platz 3: depth 12	1	9	—	2
b 16-20-7-11	Platz 1: depth 13	1	7	4	—
b 16-20-7-11	Platz 2: depth 12	1	9	3	—
b 16-20-7-11	Platz 3: keep 20	1	1	6	—

Tabelle 6.10: Vergleich der Strategien für Queens. Nur wenige Strategien schneiden bei diesem Benchmark überhaupt gut ab, was sich in den schlechten Werten der zweit- und drittbesten Strategien wiederspiegelt. Depth 4, das für die Problemgröße 11 gute Leistung bietet, erreicht bei Größe 10 nur einen kleinen Bruchteil der möglichen Beschleunigung. Die beste Strategie für jedes Problem ist Depth 1, die dementsprechend überall optimal abschneidet.

Problem	Beste Strategien	10	11	12
10	Platz 1: depth 1	—	1	1
10	Platz 2: depth 3	—	1	2
10	Platz 3: never 0	—	7	13
11	Platz 1: depth 1	1	—	1
11	Platz 2: depth 4	12	—	3
11	Platz 3: keep 2	3	—	4
12	Platz 1: depth 1	1	1	—
12	Platz 2: depth 3	2	1	—
12	Platz 3: depth 4	12	2	—

Gültigkeit hat. Auf eine möglichst hohe Ähnlichkeit der Eingabedaten muß bei den parallelen Läufen also nicht allzu sehr geachtet werden.

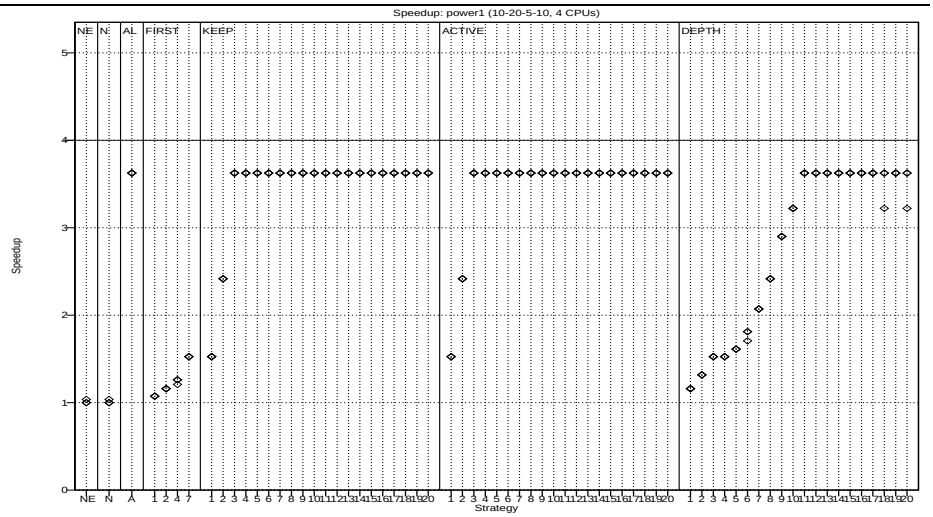
6.2.6 Effizienz durch geeignete Auswahl der Hierarchieebenen

Einige Benchmarks wie Power und Barnes haben verschiedene Hierarchieebenen der Rekursion, wie bei der Beschreibung der Benchmarks erläutert wurde. Für die effiziente Parallelisierung ist entscheidend, daß nur rekursive Prozeduren der oberen Hierarchieebenen parallelisiert werden, damit die Granularität der Teilprobleme nicht zu fein wird — bei Barnes Hut mit 16 384 Körpern ist z.B. die Berechnungszeit für ein Blatt der oberen rekursiven Prozedur `computesubtree` 1,59 ms, während die darunterliegende Prozedur `walksub` nur 0,01 ms dauert, also zwei Größenordnungen feingranularer ist.

Abbildung 6.14 stellt die Beschleunigungen des Power Benchmarks bei identischer Problemgröße gegenüber: Im ersten Fall wurde nur die oberste Prozedur `feeder` parallelisiert, was eine sehr gute Beschleunigung von 3,7 ergibt, bei der viele Strategien optimale Ergebnisse bringen. Im zweiten Fall wurde zusätzlich die `lateral` Ebene parallelisiert, was die Granularität stark verfeinert. Dementsprechend sinkt die Beschleunigung auf maximal 3,3 und die genaue Strategiewahl wird viel kritischer, was die Auswirkung einer nicht perfekten Strategiewahl negativ verstärkt. Wie in Tabelle 6.2 gezeigt, sinkt dabei der Prozentsatz der Strategien, die mindestens 80% der (bereits schlechteren) Beschleunigung erreichen, von 70% auf 7% bzw. 50%.

Um diesen Leistungszuwachs auszunutzen, gibt das in Abschnitt 5.6.3.6 beschriebene REAPAR Werkzeug `hierarchies_check_simulation` dem Benutzer Hinweise, welche Prozeduren von einer Noparallel-Annotation profitieren könnten. Wie dort gezeigt, empfiehlt dieses Hilfsprogramm für Barnes Hut und Power genau die Annotationen, die in der Realität zum Erfolg führen.

Nur oberste Ebene parallelisiert:



Oberste beide Ebenen parallelisiert:

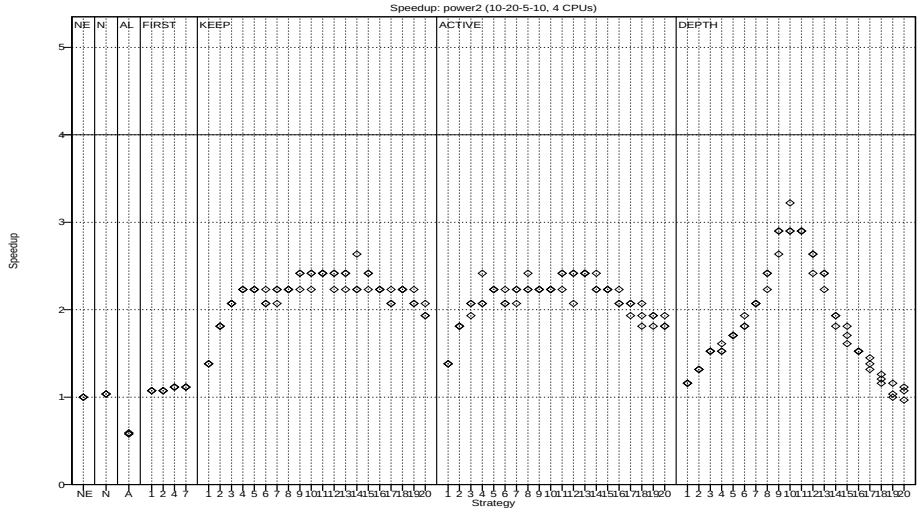


Abbildung 6.14: Beschleunigungen von Power 10 20 5 10 in Abhängigkeit von den parallelisierten Hierarchieebenen: Gute Ergebnisse mit sehr unkritischer Strategiewahl (sogar das triviale Always liefert optimale Beschleunigungen) bzw. schlechtere Beschleunigungen mit nur wenigen guten Strategien.

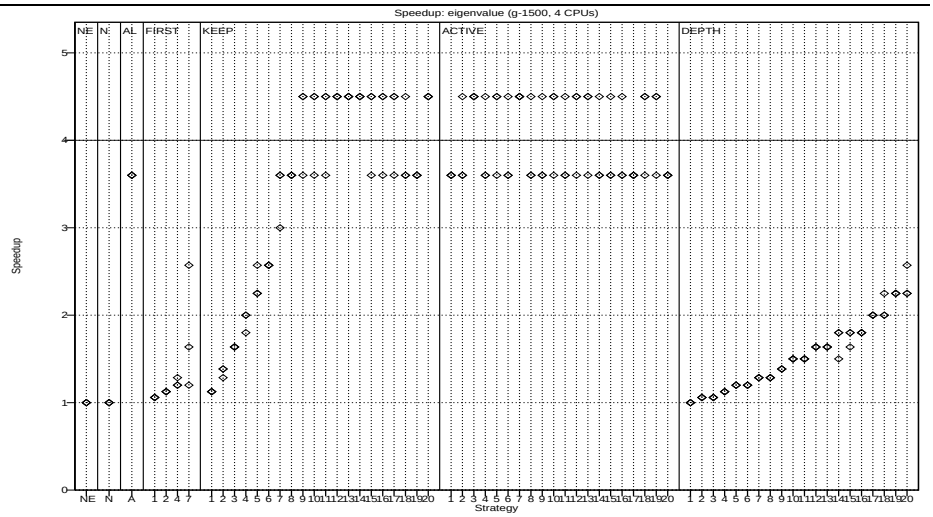
6.2.7 Effizienzgewinn durch Nothread-Annotation

In Abschnitt 5.3 wurde erwähnt, daß die Nothread Annotation die Leistung von Programmen mit festem Rekursionsgrad steigern kann, indem die Threaderzeugung für den letzten rekursiven Aufruf eingespart wird.

Der Vergleich der Beschleunigungen des Eigenvalue-Benchmarks für geometrische Eigenwertverteilung in Abbildung 6.15 belegt diese Aussage: Mit Nothread-Annotation wird eine Beschleunigung von 4,5 erreicht, ohne die Annotation sinkt die maximale Beschleunigung auf 3,6 ab. Bei einer uniformen Eigenwertverteilung hat die Annotation hingegen keine Auswirkung auf die Höchstbeschleunigung, aber die Beschleunigung von einzelnen Strategien geht zurück (Always nur 3,6 statt 3,9), und die maximalen Threadzahlen verdoppeln sich.

Beim Fractal-Benchmark ergeben sich ebenfalls keine Einbußen der Maximalbeschleunigung, aber die Zahl von Strategien, für die sie erreicht wird, nimmt ab, und die Gesamtthreadzahlen steigen an, was die Strategiewahl kritischer macht.

Mit Annotation:



Ohne Annotation:

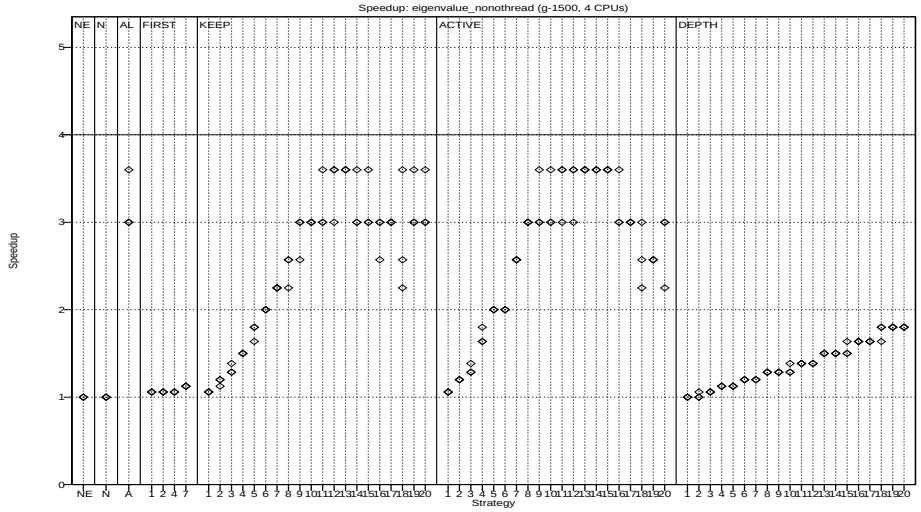


Abbildung 6.15: Beschleunigungen von Eigenvalue mit und ohne Verwendung der Nothread-Annotation bei geometrischer Eigenwertverteilung.

6.2.8 Kosten der Datensammlung

Eine Hauptforderung an die Datensammlungskomponente des Systems war, daß sie selbst den Programmablauf nur unwesentlich verzögern soll, um die Meßergebnisse nicht zu verfälschen.

Zum Nachweis, daß diese Forderung erfüllt ist, wurde die sequentielle Laufzeit der Benchmarks in ihrer Urform verglichen mit der Laufzeit der automatisch instrumentierten Benchmarks. Letztere beinhaltet auch die Ausgabe des Laufzeitprofils. Außerdem wurde auch der Laufzeitmehraufwand für die optionale vollständige Aufzeichnung des Rekursionsbaums gemessen. Tabelle 6.11 zeigt die Ergebnisse basierend auf Benchmark-Laufzeiten von ca. einer Minute.

Erwartungsgemäß fällt die Instrumentierung bei feinkörnigeren Benchmarks wie Barnes Hut stärker ins Gewicht. Sie verursacht aber höchstens 4,6% mehr Rechenzeit, typischerweise 0-2%. Diese Zahlen sind nicht prohibitiv, zumal sie sich bei der parallelen Ausführung auf die einzelnen Prozessoren aufteilen. Die Speicherkosten liegen bei vernachlässigbaren 4kBytes pro rekursiver Prozedur (unter Verwendung der Vorgabewerte Rekursionstiefe=90, Verzweigungsgrad=12).

Die Kosten der Baumaufzeichnung schwanken stark mit der Zahl der Knoten im Rekursi-

Tabelle 6.11: Prozentualer Laufzeitmehraufwand durch die Instrumentierung bezogen auf die sequentielle uninstrumentierte Laufzeit für alle Benchmarks. Beim mit (*) markierten Lauf wurde die Statistiksammlung für die untere Rekursionshierarchie per Annotation abgeschaltet.

Benchmark	Eingabe-größe	Laufzeitzuwachs instrumentiert	Laufzeitzuwachs mit Baumaufzeichnung
Barnes Hut	2048	4,6% 0,6%	258,6% 3,7% (*)
Eigenvalue	u-1500	0,1%	1,9%
Fractal	2048 2048	1,8%	8,9%
Power	10 20 5 10	1,1%	13,0%
Queens	11	0,7%	76,1%

onsbaum. Barnes Hut und Queens, deren Bäume laut Tabelle 6.1 um zwei Größenordnungen umfangreicher sind als die der anderen Benchmarks, werden deutlich langsamer. Bei diesen beiden Programmen wird auch der Speicherbedarf für den Rekursionsbaum deutlich höher, z.B. 160MB bei Barnes Hut. Die anderen Benchmarks erleiden generell mit 1-13% nur geringe Leistungseinbußen.

Abhilfe läßt sich in vielen Fällen durch die Verwendung der Nostats-Annotation schaffen, die die Datensammlung für eine Prozedur gezielt abschaltet, oder mit der Option `-autonostats` der Parallelisierung, die dann nur Daten für parallelisierte Prozeduren sammelt. Ein Abschalten der Datensammlung ist sinnvoll z.B. bei uninteressanten rekursiven Hilfsprozeduren oder für mehrstufige Rekursionshierarchien wie bei Barnes, wo nur die oberste Ebene wichtig ist. Dementsprechend schrumpft der oben erwähnte Rekursionsbaum bei Barnes Hut von 160MB auf handhabbare 5MB, die auch kein Problem mehr für die automatische Strategiewahl darstellen. Dabei sinken auch die Laufzeitkosten dramatisch von über 250% auf unter 4% ab.

Die Profil-Instrumentierung bedeutet also solch einen geringen Mehraufwand, daß sie auch für parallele Programmläufe ohne merkliche Verlangsamung aktiviert sein kann. Bei der Baumaufzeichnung muß die zu erwartende Zahl von Knoten im Rekursionsbaum berücksichtigt werden, um die Datenmenge und die Laufzeitverschlechterung in akzeptablen Grenzen zu halten. In der Praxis sind feingranulare Benchmarks mit Bäumen über 500 000 Blättern schlecht für die Aufzeichnung geeignet.

6.2.9 Aufwand der Quellcodetransformationen

Für den praktischen Einsatz des Systems muß sich der Zeitaufwand für Instrumentierung und Parallelisierung in für den Anwender akzeptablen Grenzen halten — eine Parallelisierung, die eine Stunde dauert, wird bei einer Laufzeit des fertigen Programms von zehn Minuten nicht die Gunst der Anwender finden, auch wenn sich die einmalige Parallelisierung bei mehreren Programmläufen schnell amortisiert.

Daher zeigt Tabelle 6.12 die Laufzeiten der Instrumentierung und Parallelisierung für die verwendeten Benchmarks. Zugrundegelegt wurde die verbrauchte Benutzerzeit, da der Zugriff auf die Festplatten mit Programmen und Quellcode über Ethernet erfolgte (die

Wallclock-Laufzeiten liegen ca. 15% über den Benutzerzeiten). Mit aufgeführt ist die Größe des Benchmarks in LOC sowie die Zahl der Prozeduren und der rekursiven Prozeduren, die die Quellcodetransformation entscheidend beeinflußt.

Tabelle 6.12: Dauer der Instrumentierung und Parallelisierung für die einzelnen Benchmarks.

Benchmark	LOC	Prozeduren	Rekursionen	Instrumentierung (s)	Parallelisierung (s)
Barnes Hut	2 450	42	7	28,0	63,8
Eigenvalue	550	11	1	2,0	9,3
Fractal	650	7	1	1,7	6,8
Power	1 200	24	6	9,0	33,1
Queens	450	5	2	1,2	4,7

Man sieht, daß selbst die längste Parallelisierung deutlich unter zwei Minuten dauert. Die typischen Laufzeiten liegen im Sekundenbereich und stellen damit keinerlei Akzeptanzproblem dar.

6.2.10 Aufwand der Strategiewahl

Eine Anforderung an die Strategiewahl war, daß sie selbst deutlich weniger Rechenzeit in Anspruch nehmen muß als der eigentliche Ablauf einer Berechnung. Anderenfalls könnte man auch alle sinnvoll erscheinenden Parallelisierungen einfach ausprobieren und die real beste verwenden.

6.2.10.1 Tiefenwahl-Heuristik

Tabelle 6.13 führt die Laufzeiten des `evaluate_profile` Werkzeugs auf, das für freingranulare Probleme die Tiefenstrategie wählt. Gemessen wurde die Wahl durch die AP-Heuristik, die LPS-Heuristik dauert genau so lange. Die aufgeführten Werte sind Benutzer-Sekunden, die die Zeit für die Ein/Ausgabe über NFS vernachlässigen. Die Wallclock-Laufzeiten liegen maximal 30% darüber.

Offensichtlich ist die Heuristik so schnell, daß sich ihre Laufzeit am Rande der Meßgenauigkeit von 1/50s bewegt. Dieses Ergebnis war zu erwarten, da die Tiefenstrategiewahl lediglich Additionen und Durchschnittsbildungen auf der Tabelle des Laufzeitprofils durchführt und die Erfüllung der Heuristikbedingungen mit wenigen Maschinenbefehlen geprüft werden kann.

6.2.10.2 Simulation

Die Simulation ist naturgemäß aufwendiger als die Tiefenheuristik, da sie mehr Rechenaufwand für den virtuellen Threadablauf benötigt und die Datenmengen weitaus höher sind — ein Rekursionsbaum nur der obersten Hierarchieebene von Barnes Hut ist bei Eingabegröße von 131 072 über 590kB groß, im Gegensatz zu 15kB für das Laufzeitprofil. In Tabelle 6.14 ist die Dauer der Strategiewahl in Benutzersekunden für die grobgranularen Benchmarks angegeben. Die Wallclock-Laufzeit ist wegen des hohen Datenaufkommens und des langsamen

Tabelle 6.13: Laufzeiten der AP-Tiefenheuristik für die feingranularen Benchmarks.

Benchmark	Problem-größe	Laufzeit der Strategiewahl (s)
Fractal	1 $1\,024^2$	0,01
	1 $4\,096^2$	0,04
	2 $1\,024^2$	0,01
	2 $4\,096^2$	0,02
	3 $1\,024^2$	0,03
	3 $4\,096^2$	0,03
	4 $1\,024^2$	0,02
	4 $4\,096^2$	0,02
	5 $1\,024^2$	0,02
	5 $4\,096^2$	0,04
Queens	10	0,03
	11	0,05
	12	0,06

NFS-Zugriffs wieder etwas höher. Für jede Strategieklasse wurden die Zeiten der Simulationen bis einschließlich der Simulation addiert, die die Auswahlheuristik als Empfehlung ausgibt. Die Zahl dieser Simulationsläufe ist in der Tabelle in Klammern angegeben und entspricht dem gewählten Strategieparameter in Tabelle 6.5. Läufe, bei denen keine erfolgversprechende Strategie gefunden wurde, sind mit (*) markiert und zeigen die Summe der Gesamtzeit für die erfolglosen Simulationen. Es wurden Strategieparameterwerte von 1 bis 10 verwendet.

Mit dem Parameter der endgültig gewählten Strategie, also der Zahl der Simulationsläufe, erhöht sich die Laufzeit der Strategiewahl. Maximal ist sie bei Strategieklassen, die für das gegebene Problem keine erfolgversprechende Simulation bieten. Dies ist besonders bei Eigenvalue mit geometrischer Verteilung der Fall. Außerdem wächst der Simulationsaufwand mit der Größe des Rekursionsbaums, wie der Vergleich zwischen Eigenvalue u-1 500 und u-9 000 bei fast identischer Strategiewahl zeigt.

Die Gesamtlaufzeit der Strategiewahl, d.h. die Summe über alle Strategieklassen, liegt bei fast allen Benchmarks zwischen 5 und 20 Sekunden, also deutlich unter der eigentlichen Laufzeit eines Benchmarks. Bei den größeren Barnes Hut Problemen dauert die Strategiewahl über zwei Minuten, aber die Laufzeit des Problems selbst liegt nochmal um den Faktor 25 darüber. Zudem kann nach Abschnitt 6.2.5 die einmal gewählte Strategie für eine Vielzahl von Problemen verwendet werden.

6.2.11 Kombination von Strategien

In Abschnitt 4.3.1.2 wurde die Möglichkeit erwähnt, die allgemeinen Strategien mit den Tiefenstrategien zu verbinden, z.B. also die Strategie „Erzeuge Threads bis zur Tiefe t, aber nur wenn aktuell weniger als n Threads aktiv sind“.

Diese kombinierten Strategien wurden im Vorfeld der Arbeit mit Handparallelisierung eingehend untersucht. Bei keinem der Benchmarks ergab sich durch die Strategiekombi-

Tabelle 6.14: Laufzeiten der Simulationen für die grobgranularen Benchmarks und Gesamtlaufzeit für alle Strategieklassen„ Zahl der Simulationsläufe bis zur Wahl der Strategie in Klammern.

Benchmark	Problem-	Simulationslaufzeit (s)			
	größe	Keep	Active	Depth	Summe
Barnes Hut	1 024	4,85 (5)	1,68 (2)	2,89 (3)	9,42
	16 384	16,23 (8)	2,59 (1)	4,99 (3)	23,81
	131 072	90,34 (9)	13,32 (1)	36,97 (4)	140,63
Eigenvalue	u-1 500	1,64 (2)	0,92 (1)	2,67 (3)	5,23
	g-1 500	8,55 (*)	1,77 (2)	8,49 (*)	18,81
	u-9 000	2,67 (2)	2,63 (2)	3,97 (4)	9,47
	g-9 000	8,64 (*)	1,92 (2)	9,10 (*)	19,66
	u-36 000	6,14 (2)	5,66 (2)	12,69 (4)	24,49
	g-36 000	8,82 (*)	1,85 (2)	9,45 (*)	20,12
Power	10 20 5 10	2,98 (3)	2,69 (3)	7,93 (9)	13,60
	16 20 7 11	1,78 (2)	1,74 (2)	6,21 (8)	9,73

nation eine zusätzliche Beschleunigung; es wurde lediglich der Parameterbereich für gute Beschleunigungen vergrößert. Außerdem schneiden feingranulare Benchmarks schlechter ab, da in jedem Fall die Threadzahl unter Verwendung von kritischen Abschnitten mitgeführt werden muß.

Da ohne Leistungsvorteile die zusätzliche Komplexität und die Vergrößerung des Parameterraums nicht lohnend erschienen, wurde die Kombination von Strategien nicht weiter verfolgt.

6.3 Benchmarkläufe mit mehr Prozessoren

Kurz vor Abschluß der Arbeit bot sich die Gelegenheit, einige der Benchmarks auf Rechnern mit mehr als vier Prozessoren durchzumessen. Von der Firma Sun Microsystems Deutschland wurden dazu freundlicherweise zwei Sun Enterprise Server zur Verfügung gestellt:[2] eine Sun E4000 mit 7,5GB Hauptspeicher, 12 UltraSPARC-II 250MHz Prozessoren mit je 4MB Cache sowie eine Sun E6000 mit 8GB und 30 250MHz UltraSPARC-II Prozessoren mit je 1MB Cache, beide unter Solaris 5.5.1.

Es wurden alle Probleme durchgemessen, die auf den ursprünglichen 100MHz HyperSPARC Prozessoren eine Laufzeit von mindestens 18 Sekunden hatten. Leider sind die Ergebnisse teilweise nicht repräsentativ, da die sequentiellen Laufzeiten vieler Benchmarks durch die stärkeren Prozessoren und den größeren Cache der Testmaschinen auf 1-10 Sekunden gedrückt wurden. Bei einer Meßgenauigkeit der Auswertung von einer Sekunde sind die entsprechenden Beschleunigungsaussagen bedeutungslos. Dennoch ergaben sich wertvolle Informationen über die Skalierbarkeit, die unten diskutiert werden.

[2]An dieser Stelle möchte ich ganz herzlich Peter Hausdorf und Ulrich Graef vom Benchmark Center Germany sowie Peter Möller und Franz Haberhauer von Sun für die Bereitstellung der Rechner und die Durchführung der Versuche und bei Heinz Herrmann für sein erfolgreiches Lobbying bei Sun bedanken.

6.3.1 Beschleunigungen

Tabelle 6.15 faßt die Ergebnisse der längeren Testläufe zusammen. Um die Rechenzeit nicht übermäßig zu verlängern, wurden nur zwei statt drei Läufe pro Strategie durchgeführt und nur Strategieparameter von Keep und Active nur von 1 bis 10 statt von 1 bis 20 gemessen. Außerdem wurden nur Fractal-Ausschnitt 1 und 4 betrachtet.

Tabelle 6.15: Beste erreichte Beschleunigung und entsprechende Strategie für alle Benchmarks auf 8, 12 und 30 Prozessoren. Die mit „–“ markierten Messungen konnten aus Zeitgründen nicht durchgeführt werden.

Benchmark-name	Problem-größe	8 CPUs		12 CPUs		30 CPUs	
		Bes.	Strategie	Bes.	Strategie	Bes.	Strategie
Barnes Hut	131 072	4,964	Depth 5	5,900	Depth 5	8,176	Depth 5
Eigenvalue	u-9 000	8,094	Active 9	12,381	Keep 4	30,706	Active 2
	g-9 000	8,250	Active 1	16,500	Active 2	33,000	Active 1
	u-36 000	-,-	— -	12,008	Active 8	30,197	Active 9
	g-36 000	-,-	— -	12,556	Active 2	28,750	Active 2
Fractal	1 4 096-4 096	5,333	Depth 3	8,000	Depth 4	8,000	Depth 3
	4 4 096-4 096	7,273	Depth 4	10,000	Depth 3	20,000	Depth 4
Power	16-20-7-11	7,313	Active 4	7,313	Keep 3	9,750	Active 1
Queens	12	4,250	Depth 1	3,400	Depth 1	2,833	Depth 1

Zur genaueren Betrachtung der einzelnen Benchmarks und ihres Abschneidens:

Barnes Hut skaliert nicht linear, aber erreicht mit zunehmender Prozessorzahl auch zunehmende Beschleunigung. Dieses Verhalten liegt vermutlich an der komplexen Interaktion der simulierten Partikel mit vielen parallelen Lesezugriffen auf gemeinsame Informationen. Cilk berichtet für diesen Benchmark auf 8 Prozessoren auch nur eine Beschleunigung von 4,98. Olden erreicht hingegen mit dem handoptimierten Benchmark für diese Prozessorzahl 5,29, also 6,6% mehr, auf 16 CPUs 8,13 und auf 32 CPUs 11,20 (skaliert auf 30 CPUs 10,50, d.h. 28% besser als REAPAR).

Eigenvalue zeigt auch bei 30 Prozessoren noch eine perfekte Beschleunigung, sowohl für uniforme als auch für die irreguläre geometrische Eigenwertverteilung. Begründet ist das in der hohen Datenlokalität des Programms, dessen Teilprobleme völlig unabhängig voneinander auf eigenen Daten rechnen, und in der groben Granularität des Problems. Die überlinearen Beschleunigungen lassen sich durch Lokalität und Cache erklären.

Fractal erreicht abhängig vom Bildausschnitt sehr gute Beschleunigungen — Ausschnitt 4 bietet überall etwa gleich viel Arbeit. Daß die Beschleunigungen nicht linear sind, liegt auch an der Länge eines einzelnen Laufs von nur 16 Sekunden bei Ausschnitt 1.

Power bietet bei 8 Prozessoren noch eine hervorragende Beschleunigung. Das schechte Abschneiden bei höherer Prozessorzahl liegt daran, daß nur die oberste Rekursionshierarchie parallelisiert wurde, die im untersuchten Problem lediglich 12 Knoten enthält, also nicht genügend Parallelitätspotential für mehr Prozessoren bietet. Die aggressivere Handparallelisierung von Olden erreicht auf 8 Prozessoren nur eine Beschleunigung von 6,92, auf 16 Prozessoren hingegen schon 14,85 und auf 32 CPUs beachtliche 27,50.

Queens enttäuscht mit einer zunehmend schlechten Beschleunigung bei wachsender Prozessorzahl. Vermutlich läuft die produktive Berechnung in diesem feingranularen Benchmark auf den modernen Prozessoren so schnell ab, daß ein Großteil der Laufzeit für die Threadbehandlung verwendet wird. Dies wird durch das Verhältnis von 2:1 Benutzer:Systemzeit bei selbst den schnellsten Strategien bestätigt. Außerdem ist die sequentielle Laufzeit von Queens 12 hier nur 17 Sekunden. Ein direkter Vergleich mit den Cilk-Ergebnissen ist nicht möglich, da die Cilk-Version indeterministisch ist, aber Cilk ist von seinem Nanoscheduling her für solch feingranulare Probleme wohl besser vorbereitet.

Zusammenfassend läßt sich sagen, daß die Parallelisierung durch REAPAR für grobgranulare Probleme auch für hohe Prozessorzahlen perfekte Ergebnisse zeigt. Bei sehr feingranularen Problemen und solchen, die nicht genügend Parallelitätspotential bieten, sind die Beschleunigungen weniger zufriedenstellend, aber in jedem Fall wird eine reale Beschleunigung erreicht.

Für die ursprüngliche Zielarchitektur von REAPAR, d.h. dem Desktoprechner mit 2-8 CPUs, auf dem ein unerfahrener Benutzer eine substantielle Beschleunigung ohne eigenes Zutun erzielen möchte, sind die Ergebnisse mehr als zufriedenstellend.

6.3.2 Strategiewahl

Um die Skalierung der Strategiewahlkomponente zu testen, wurden für die 8, 12 und 30 CPUs sie entsprechenden Simulationen bzw. Heuristiken zur Strategiewahl durchgeführt. Die Ergebnisse der Simulationen für die grobgranularen Benchmarks finden sich in Tabelle 6.17, die der Heuristiken für die feingranularen Programme in Tabelle 6.16.

Diese Abschnitte gehören konzeptionell bereits zur Validierung, da die Versuche nach Abschluß aller Arbeiten am System durchgeführt wurden. Sie sind dennoch in diesem Kapitel aufgeführt, weil sie thematisch zu den Experimenten mit höherer Prozessorzahl passen.

6.3.2.1 Heuristik

Tabelle 6.16: Durch die AP-Tiefenheuristik gewählte Strategien für die feingranularen Benchmarks und ihr reales Abschneiden.

CPU-Zahl	Benchmark	Problemgröße	Beste reale Strategie	Gewählte Strategie	Erreichte Leistung
8	Fractal	$1\,4096^2$	Depth 3	Depth 2	73%
		$4\,4096^2$	Depth 4	Depth 2	75%
	Queens	12	Depth 1	Depth 1	100%
12	Fractal	$1\,4096^2$	Depth 4	Depth 2	50%
		$4\,4096^2$	Depth 3	Depth 2	67%
	Queens	12	Depth 1	Depth 2	100%
30	Fractal	$1\,4096^2$	Depth 3	Depth 3	100%
		$4\,4096^2$	Depth 4	Depth 3	80%
	Queens	12	Depth 1	Depth 2	32%

Die AP-Tiefenheuristik wählt für einen Großteil der Probleme auf 8 bis 30 CPUs eine gute Tiefe, oft sogar das Optimum. Das schlechte Abschneiden bei Queens liegt mit daran, daß dieser Benchmark wie oben beschrieben mit zunehmender Prozessorzahl schlechtere Leistungen erbringt. Die Tiefe skaliert mit der Zahl der Prozessoren, da diese direkt in die Heuristik eingeht.

6.3.2.2 Simulation

Tabelle 6.17: Von der Simulation gewählte Strategien für 8, 12 und 30 Prozessoren und ihr reales Abschneiden für die grobgranularen Benchmarks. Bei mit „—“ markierten Einträgen empfahl das System keine Strategie.

CPU-Zahl	Benchmark	Problemgröße	Beste reale Strategie	Gewählte Strategie 1. Klasse	2. Klasse	3. Klasse
8	Barnes Hut	131 072	Depth 5	Depth 5 100%	Keep 7 50%	Active 1 59%
	Eigenvalue	u-9 000	Active 1	Depth 5 89%	Active 2 98%	Keep 9 98%
		g-9 000	Active 9	Active 2 100%	Keep —	Depth —
	Power	16-20-7-11	Active 4	Keep —	Depth —	Active —
12	Barnes Hut	131 072	Depth 5	Depth 5 100%	Active 2 22%	Keep —
	Eigenvalue	u-9 000	Keep 4	Active 2 98%	Depth 5 86%	Keep —
		g-9 000	Active 2	Active 2 100%	Keep —	Depth —
		u-36 000	Active 8	Active 1 93%	Depth 5 76%	Keep —
		g-36 000	Active 2	Active 2 100%	Keep —	Depth —
	Power	16-20-7-11	Keep 3	Keep —	Depth —	Active —
30	Barnes Hut	131 072	Depth 5	Active 1 11%	Depth 5 100%	Keep —
	Eigenvalue	u-9 000	Active 2	Active 2 100%	Depth 6 85%	Keep —
		g-9 000	Active 1	Active —	Keep —	Depth —
		u-36 000	Active 9	Active 1 93%	Depth 6 78%	Keep —
		g-36 000	Active 2	Active —	Keep —	Depth —
	Power	16-20-7-11	Active 1	Keep —	Depth —	Active —

Bei den Simulationen fällt auf, daß die Qualität der Strategiewahl durchgehend sehr gut ist und maximal 11% vom Optimum entfernt liegt, vom Ausreißer bei Barnes Hut auf 30 CPUs abgesehen.

Für Power wird gar keine Strategie empfohlen, da die Simulationen nur eine geringe Auslastung der Maschine und ein kleines Parallelitätspotential vorhersagen — und tatsächlich nutzt Power in der Praxis den Rechner nicht aus und skaliert schlecht in der Leistung. Bei 30 CPUs wird auch für die geometrische Eigenwertverteilung keine Strategie mehr empfohlen, weil die Filter-Heuristik aktiv wird, die das Verhältnis von Knoten im größten Unterbaum zu Knoten im Gesamtbaum auf maximal $1/$*Zahl der CPUs* begrenzt. Ohne dieses Filter würde Active 1 empfohlen, was 80% bzw. 100% der optimalen Leistung auf 12 bzw. 30 CPUs bringt.

Die Simulation spiegelt also auch bei höheren Prozessorzahlen das Verhalten einer Parallelisierungsstrategie gut wieder, wobei für Prozessorzahlen ab 30 offenbar eine weniger strikte Filterheuristik die Zahl der überhaupt empfohlenen Strategieklassen sinnvoll erhöhen würde.

Zusammenfassend läßt sich sagen, daß sowohl die erzielten Beschleunigungen als auch die Strategiewahl im Bereich von 8 bis 30 Prozessoren gut skalieren, obwohl das System nur auf 4 Prozessoren entwickelt und getestet wurde. Damit ist auch der Beweis erbracht, daß keine Optimierung gezielt auf 4 Prozessoren stattfand, sondern allgemeingültige Methoden und Heuristiken entwickelt wurden.

6.4 Fallstudie

Zur weiteren Untermauerung der Arbeit wurde im Sommersemester 1997 im Praktikum „Speichergekoppelte Multiprozessoren“ eine Fallstudie durchgeführt, die die Leistung des Systems mit den Ergebnissen einer Handparallelisierung vergleicht.[3]

Die Praktikumsteilnehmer hatten zum Zeitpunkt des Experiments bereits Erfahrungen mit der Programmierung von Mehrprozessorrechnern mit gemeinsamem Speicher gesammelt. Als Ausgangspunkt wurde der rekursive Algorithmus des Fractal-Benchmarks vorgestellt und seine Arbeitsweise zum besseren Verständnis unter X11 grafisch visualisiert — die gute Visualisierbarkeit des Benchmarks sowie die nicht perfekte Beschleunigung durch das REAPAR System gaben den Ausschlag zu dieser Wahl. Grundlegende Möglichkeiten der Parallelisierung wurden aufgezeigt, wobei die Teilnehmer angehalten waren, auch eigene Ideen einzubringen. Der gut kommentierte sequentielle Quellcode des Benchmarks diente als Ausgangspunkt der Parallelisierung.

Während der nächsten zwei Praktikumswochen parallelisierten die Teilnehmer das Programm nach ihren eigenen Vorstellungen. Dabei hielt die Infrastruktur die erfolgten Übersetzer- und Programmläufe fest. Als Abschluß des Versuchs wurden die erzielten Beschleunigungen für verschiedene Problemgrößen gemessen. In einem Fragebogen wurden die Teilnehmer über ihren Programmier-Hintergrund und ihre subjektive Einschätzung des Zeitaufwands befragt. Außerdem wurden die protokollierten Entwicklungszeiten ausgewertet, die Zahl der hinzugefügten Programmzeilen gemessen und die erstellten Programme analysiert.

Als Rechner diente wie bei den 4-Prozessor Ergebnissen dieses Kapitels auch eine SPARC-Station 20 mit 4 100MHz HyperSPARC-Prozessoren, d.h. die Testumgebung ist identisch. Tabelle 6.18 gibt einen Überblick der Ergebnisse.

Es fällt auf, daß alle Teilnehmer ein Warteschlangenkonzept für die Abarbeitung der anfallenden Bildausschnitte wählten — vermutlich, weil im Praktikum die bisherigen Aufgaben mit einer festen Gruppe von Threads und entsprechenden Warteschlangen für Arbeitsaufträge gelöst wurden. Einige Teilnehmer führen Optimierungen hinsichtlich der kleinsten

[3] An dieser Stelle möchte ich dem Institut für Rechnerentwurf und Fehlertoleranz und insbesondere Winfried Grünewald danken, der durch die Bereitstellung von Praktikumszeit und seine generelle Unterstützung das Experiment ermöglichte.

Tabelle 6.18: Ergebnisse der Fallstudie mit Praktikumsteilnehmern, Beschleunigungen für das Ur-Fraktal mit 1024×1024 Bildpunkten.

	Teilnehmer 1	Teilnehmer 2	Teilnehmer 3	Teilnehmer 4[a]
Parallelisierungsstrategie	Globale Arbeits-Warteschlange, doppelt geschachtelte Threaderzeugung, feste maximale Threadzahl, Threads arbeiten Schlange ab und erzeugen dabei neue Einträge, Mindestgröße des Bildausschnitts für Threaderzeugung	Globale Arbeits-Warteschlange, zwei Client / Server-Prozeduren (a, b), die die Schlange abarbeiten und teilweise neue Arbeit einreihen, Hauptprogramm startet Mischung aus a und b, feste Threadzahl[b]	Globale Arbeits-Warteschlange, je nach Größe des Bildausschnitts werden dynamisch 1-3 Arbeitseinheiten erzeugt, Hauptprogramm startet feste Anzahl von Threads, die Schlange abarbeiten	Globale Arbeits-Warteschlange, Bildausschnitte werden nur bis zu beim Programmstart bestimmbarer Tiefe und Größe in Schlange eingereiht, Wiederverwendung von Threads
Beschleunigung	3,50	2,97	3,52	3,76
Hinzugefügte Zeilen/kB	714 / 16kB	106 / 2,7kB	141 / 3,3kb[c]	334 / 6,9kB[d]
Entwicklungszeit	15h	5h	3h	2,5h
Prog.-Erfahrung	4 Jahre	8 Jahre	8 Jahre	14 Jahre
Semester	8	10	8	12

[a] Tutor, außer Konkurrenz da weitgehendes Vorwissen und Verwendung von C++ Threadklassen.

[b] Vermutlich werden durch Programmierfehler Bereiche mehrfach berechnet, was die schlechte Beschleunigung erklärt.

[c] Verwendet zusätzlich vom Praktikum her bereits vorhandene Datenstrukturen

[d] Alles in C++ umformuliert und dann erweitert

einzureihenden Arbeitseinheit oder der Rekursionstiefe ein. Die nötige Arbeit zur Parallelisierung und Fehlersuche nahm einige Stunden bis mehrere Arbeitstage in Anspruch.

Zum Vergleich: Wie vorher in diesem Kapitel aufgeführt, parallelisiert REAPAR den gegebenen Quellcode automatisch in 7 Sekunden. Der sequentielle Beispielslauf mit Profilaufzeichnung dauert 20 Sekunden, die Strategiewahl durch die AS-Heuristik weniger als eine Sekunde. Damit liefert das System in weniger als einer Minute eine Parallelisierung, die eine Beschleunigung von 3,33 auf den Testdaten erreicht, also nur 12,9% schlechter ist als die beste Handparallelisierung durch den erfahrenen Tutor, und das ohne jegliche Vorkenntnisse des Benutzers. Bis die schnellste Handparallelisierung abgeschlossen ist, kann der REAPAR-Benutzer bereits Hunderte von parallelen Programmläufen durchführen!

Kapitel 7

Validierung

Das vorangehende Ergebnis-Kapitel hat gezeigt, daß das System für die untersuchten Benchmarks die in Kapitel 4 gestellten Anforderungen erfüllt.

Um dem Anspruch auf allgemeine Gültigkeit für irreguläre rekursive Programme zu genügen, muß aber noch nachgewiesen werden, daß das System nicht gezielt auf die verwendeten Benchmarks hin optimiert wurde und nicht nur für diese anwendbar ist. Dieser Nachweis wird durch Untersuchung einer Kontrollmenge von Benchmarks geführt, die während der Systementwicklung und der Leistungsmessungen völlig unberücksichtigt blieben und sogar erst nach Abschluß der Arbeiten am System implementiert bzw. portiert wurden. Erreicht das System auch für diese neuen Benchmarks gute Ergebnisse, kann eine allgemeine Verwendbarkeit unterstellt werden.

Dieses Kapitel beschreibt die zur Validierung verwendeten Benchmarks der Kontrollmenge, diskutiert ihre Besonderheiten und zeigt die vom System erzielten Ergebnisse.

7.1 Bemerkungen

7.1.1 Skalierbarkeit

Die Skalierbarkeit von Parallelisierungsstrategien und Methoden der Strategiewahl wurde bereits im vorigen Kapitel in Abschnitt 6.3 für Maschinen mit 8 bis 30 Prozessoren nachgewiesen. Für weitere Benchmarkläufe bestand keine Möglichkeit mehr, so daß die Ergebnisse der Validierungs-Benchmarks nur von 4 Prozessoren berichten können. Abgesehen von Leistungseinbußen bei extrem feingranularen Problemen bei hohen Prozessorzahlen sind aber keine Probleme mit der Skalierbarkeit zu erwarten — die erwähnten Läufe auf den größeren Maschinen fanden lange nach Abschluß der Systemarbeiten statt, können also bereits selbst zur Validierung gezählt werden.

7.1.2 Anpassung für REAPAR

Der Zeitaufwand zur Anpassung der Programme an REAPAR war gering. Hauptsächlich mußten Konstrukte und Funktionen entfernt werden, die von Systemen der Benchmarkquellen (Cilk und Olden) stammten. Außerdem wurden, wenn nötig, Programmannotationen für REAPAR eingefügt und zu parallelisierende Prozeduren mit Rückgabewert umgeschrieben in `void` Prozeduren mit einem zusätzlichen Referenzparameter. Die Umsetzung des Heat-Benchmarks dauerte z.B. nur 36 Minuten einschließlich Fehlersuche und Test. Für den realen Einsatz von REAPAR fällt die „Aufräumphase“ der Reduktion auf sequentielles

ANSI-C ganz weg, da ja keine fremden Parallelitätskonstrukte zu entfernen sind; lediglich die Annotation und die Einführung von Referenzparametern sind in diesem Falle nötig, wobei die automatische Identifikation rekursiver Prozeduren und die Ausgabe der Aufrufrelation durch REAPAR eine zusätzliche Hilfe sind.

7.2 Bitonic Sort

Das Bitonische Sortieren [9] wurde 1986 als optimaler paralleler Sortieralgorithmus für Maschinen mit gemeinsamem Speicher vorgeschlagen. Es hat Eingang gefunden in die Benchmark-Suites mehrerer Systeme, z.B. Olden.

7.2.1 Beschreibung

Analog zur Beschreibung der Benchmarks im vorigen Kapitel folgt hier die Einordnung:

Anwendung: Datenverarbeitung — Sortieren von Zahlen.

Algorithmus: Eine Folge von Zahlen ist bitonisch, wenn sie aus einer aufsteigenden Folge gefolgt von einer absteigenden Folge besteht, oder zyklisch rotiert werden kann, so daß diese Eigenschaft zutrifft. Der Sortieralgorithmus von Batcher [6] erzeugt rekursiv bitonische Teilfolgen (`sort`) und fügt diese dann rekursiv zum sortierten Endergebnis zusammen (`merge`). Dieses Vorgehen ist Basis vieler paralleler Implementierungen, z.B. von Nicolau [9].

Parallelität: Parallelisierung durch gleichzeitiges Behandeln des linken und rechten Teilbaums sowohl in `sort` als auch in `merge`.

Irregularität: Irregulär, da die Sortierbäume und damit die Abläufe irregulär sind (die Rekursionsbäume haben für eine feste Problemgröße unabhängig von den Daten eine feste Form, die aber irregulär ist).

Der Wurzelknoten ist typischerweise 2,77-unbalanciert, d.h. recht unausgewogen. Der Gesamtbaum ist (0,92; 1,00) / (0,95; 1,5) / (1,0; 3,0)-balanciert, also sind alle Unterbäume maximal um den Faktor drei verschieden. Um mindestens 66% der Knoten abzudecken, reicht eine Unbalanciertheit von 1,0 aus.

Rekursion: Zwei 2-fach rekursive Prozeduren `sort` und `merge`, wobei `sort` nicht nur sich selbst, sondern auch `merge` aufruft.

Rekursionsbaum: In Abbildung 7.1 findet sich ein beispielhafter Rekursionsbaum.

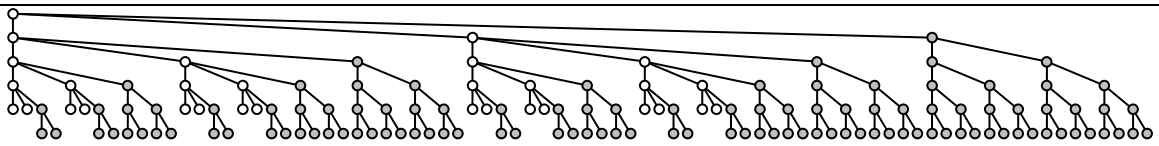


Abbildung 7.1: Bitonic Sort — Rekursionsbaum für 32 Zahlen. Die weißen Knoten gehören zur `sort`-Prozedur, die grauen zu `merge`.

Größe: ca. 400 LOC

Quelle: Abgeleitet aus einem Benchmark von Olden [16], basierend auf dem parallelen Algorithmus von Nicolau [9] und dem Ur-Algorithmus von Batcher [6].

Eingabedaten: Zufällige Zahlenfolgen der Länge n, fester Startwert des Zufallsgenerators zur Vergleichbarkeit.

Umsetzung: Aus dem Olden-Quellcode wurden alle Olden-spezifischen Datentypen und Funktionen entfernt. Der resultierende sequentielle Quellcode wurde mit Annotationen versehen, um die Berechnung von Ergebnissen vor ihrer Verwendung sicherzustellen (Needresults) und die Threaderzeugung für den jeweils linken Teilbaum zu unterbinden (Nothread). Die Parallelisierung erfolgte automatisch durch das REAPAR System.

Kenngrößen: Auf einem 100MHz HyperSPARC Prozessor ergeben sich folgende Werte:

Eingabe-größe	Lauf-zeit (s)	Blätter	Knoten	ms/ Blatt	Rek.Tiefe max/freq	Verz.Grad max/freq
16 384	2,6	212 992	196 610	0,012	14/14	3/0
131 072	37,0	2 097 152	1 966 082	0,012	17/17	3/0
524 288	164,0	9 437 184	8 912 898	0,012	19/19	3/0

Besonderheiten: Sehr feingranularer Benchmark, der daher erwartungsgemäß nur für die Tiefenstrategien gute Beschleunigungen liefert.

Während des Programmablaufs wird zuerst aufsteigend und dann absteigend sortiert, um eine eventuelle Teilsortierung der zu sortierenden Zufallszahlen auszugleichen.

Im Gegensatz zu z.B. Barnes Hut, bei dem die Konstruktion der Datenstruktur nur einen minimalen Teil der Laufzeit benötigt und daher als Noparallel annotiert wurde, lohnt sich die Parallelisierung des Baumaufbaus bei Bitonic Sort. Der zugrundeliegende Olden-Quellcode bot bereits eine für die Parallelisierung geeignete Formulierung des Baumaufbaus, die für REAPAR beibehalten wurde und ohne Änderungen im Parallelen funktioniert.

7.2.2 Beschleunigungen

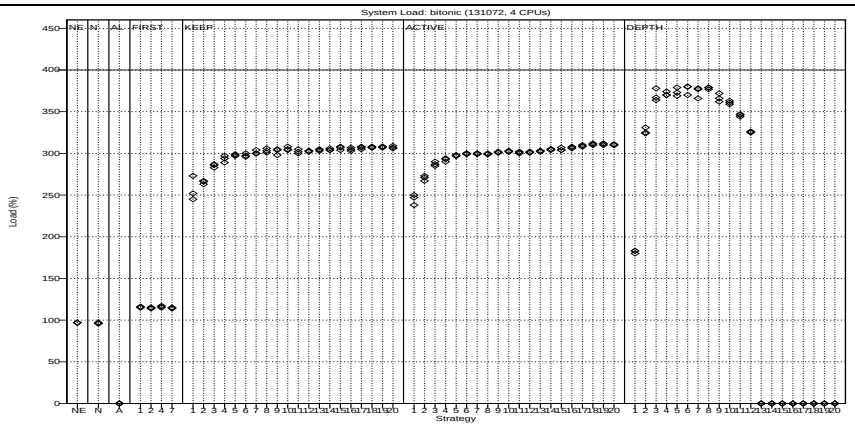
Abbildung 7.2 zeigt die für alle Parallelisierungsstrategien erreichten Beschleunigungen, Systemauslastungen, Threadzahlen und System/Benutzerzeiten bei Eingabegröße 131 072. Die Messungen für 524 288 sehen sehr ähnlich aus und werden daher nicht gesondert aufgeführt, die Messungen für 16 384 sind aufgrund der geringen sequentiellen Laufzeit von 3s nicht fein genug auflösbar.

Wie beim ebenfalls sehr feinkörnigen Queens-Benchmark der Benchmark-Suite schneiden alle Strategien sehr schlecht ab, die Zähler und damit kritische Abschnitte verwenden. Die Tiefenstrategie erreicht hingegen eine sehr gute Beschleunigung.

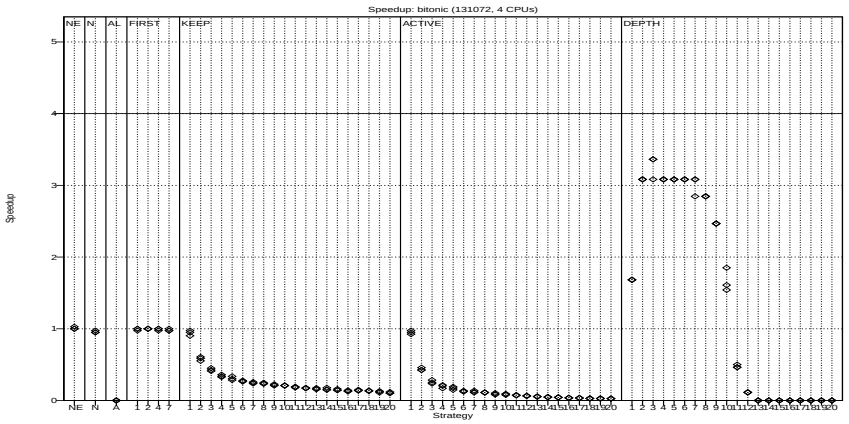
In Abschnitt 6.2.1.5 wurde bereits festgestellt, daß die Beschleunigungen durch REAPAR um 47% besser als die von Olden angegebenen Werte sind.

Die folgende Tabelle führt die besten drei Strategien und die damit erreichten Beschleunigungen auf 4 Prozessoren sowie die Prozentzahl der Strategien, die mindestens 80% der optimalen Beschleunigung erzielen, auf:

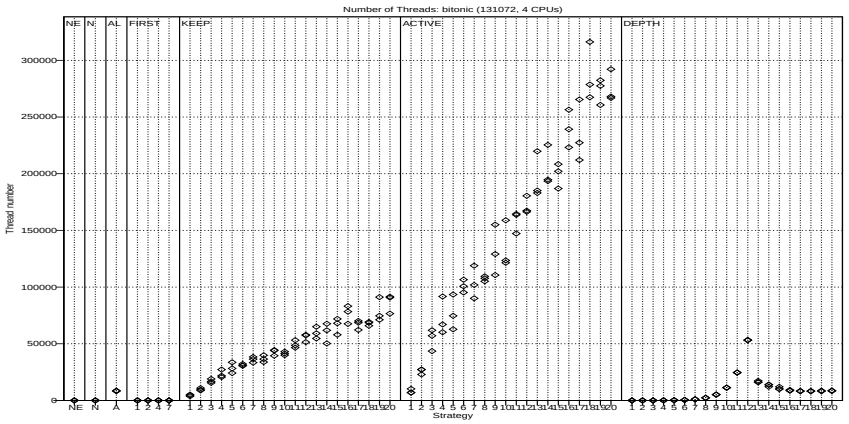
Last:



Beschleunigung:



Threadzahlen:



Zeiten:

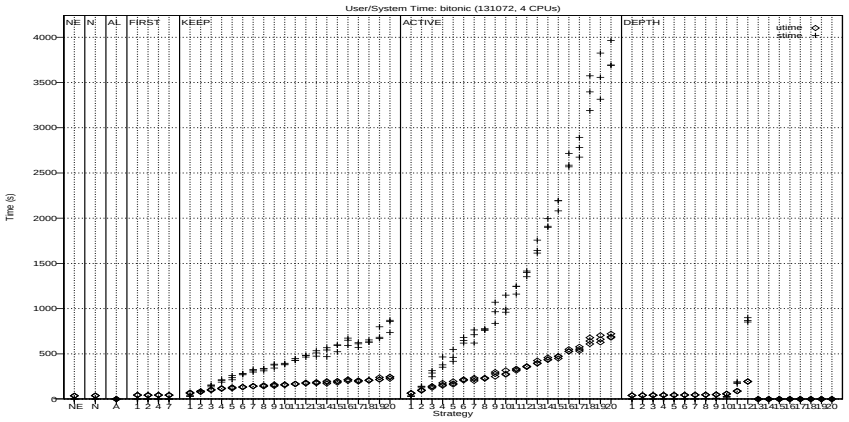


Abbildung 7.2: Bitonic Sort — Erreichte Beschleunigungen und Systemkennwerte für alle Parallelisierungsstrategien, Eingabegröße 131 072, vier Prozessoren.

Problem-	1.Platz		2.Platz		3.Platz		> 80%
größe	Bes.	Strategie	Bes.	Strategie	Bes.	Strategie	
16 384	3,000	depth 2	3,000	depth 3	3,000	depth 4	10%
131 072	3,364	depth 3	3,083	depth 2	3,083	depth 4	10%
524 288	3,154	depth 3	3,154	depth 4	3,154	depth 5	13%

7.2.3 Strategiewahl

Die AP-Tiefenheuristik wählt die folgenden Strategieparameter aus:

Problem-	Beste reale	Gewählte	Erreichte
größe	Strategie	Strategie	Leistung
16 384	Depth 2	Depth 2	100%
131 072	Depth 3	Depth 2	92%
524 288	Depth 3	Depth 2	93%

Die gewählten Strategien schneiden maximal 8% schlechter als die bestmögliche Strategie ab, d.h. die Strategiewahl wird dem Problem gerecht.

7.3 Knapsack

7.3.1 Beschreibung

Anwendung: Kombinatorische Optimierung — 0-1 Rucksackproblem, wertoptimales Füllen eines „Rucksacks“ begrenzter Kapazität mit Gegenständen, die ein vorgegebenes Gewicht und Wert haben.

Algorithmus: Rekursiver Verzweige-und-Begrenze (*branch-and-bound*) Algorithmus. Für jeden Gegenstand in absteigender Reihenfolge seines Wert:Gewicht Verhältnisses wird geprüft, ob die Lösung, bei der er in den Rucksack aufgenommen wird, besser ist als die Lösung ohne ihn. Beschränkung der Auswahl durch eine globale oberste Schranke mit der bisher besten gefundenen Lösung.

Parallelität: Parallelität durch gleichzeitiges Berechnen der Lösung mit dem aktuellen Gegenstand und der Lösung ohne ihn.

Irregularität: Irregulärer Rekursionsbaum durch die Beschneidungsgrenze.

Der Wurzelknoten ist typischerweise 10-unbalanciert, der Gesamtbaum ist beim „knap2“-Problem (0,20; 1,00) / (0,21; 1,5) / (0,41; 3,0)-unbalanciert. Um mindestens 66% der Knoten abzudecken, muß eine Unbalanciertheit von 3-5 in Kauf genommen werden.

Rekursion: Eine zweifach rekursive Prozedur.

Rekursionsbaum: Abbildung 7.3 vergleicht zwei beispielhafte Rekursionsbäume für jeweils 10 Gegenstände: Bei einem sehr ähnlichen Verhältnis zwischen Gewicht und Wert der Gegenstände kommt die Beschneidung des Baums kaum zum Zuge, während sie bei sehr unterschiedlichen Verhältnissen den Baum klein hält.

Größe: ca. 190 LOC

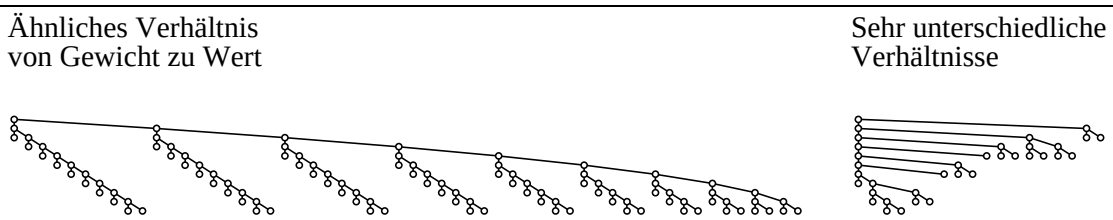


Abbildung 7.3: Knapsack — Rekursionsbaum für 10 Gegenstände, einmal mit ähnlichem Verhältnis Gewicht:Wert und entsprechend weit gefächertem Rekursionsbaum, einmal mit sehr unterschiedlichem Verhältnis und stark beschnittenem Baum.

Quelle: Abgeleitet aus einem Benchmark von Cilk [91].

Eingabedaten: Beispiels-Datensätze von Cilk, s.o.

Umsetzung: Aus dem Cilk-Quellcode wurden alle Cilk-spezifischen Datentypen und Funktionen entfernt. Die zu parallelisierende Funktion wurde so umgeschrieben, daß sie keinen Rückgabewert sondern einen Referenzparameter verwendet. Eine Needresults-Annotation vor dem Vergleich der Ergebnisse der Teilberechnungen stellt sicher, daß diese bereits vorliegen. Zur Erhöhung der Effizienz wurde der zweite rekursive Aufruf mit Nothread annotiert. Die Parallelisierung selbst erfolgte automatisch durch das REAPAR System.

Kenngrößen: Auf einem 100MHz HyperSPARC Prozessor ergeben sich für die von Cilk vorgegebenen Eingabedaten mit 30 Gegenständen folgende Werte:

Eingabe-größe	Lauf-zeit (s)	Blätter	Knoten	ms/ Blatt	Rek.Tiefe max/freq	Verz.Grad max/freq	Speicher (kb)
knap1	12	3 159 291	3 159 290	0,0038	30/25	2/0	700
knap2	472	54 732 188	54 732 187	0,0086	30/30	2/0	700

Wie der Cilk-Report anmerkt, hängt die Parallelisierbarkeit des Problems stark von den Eingabedaten ab, wie schon Abbildung 7.3 deutlich machte. Das Problem „knap1“ entspricht dem eher sequentiellen „knapsack.input“ Datensatz von Cilk, „knap2“ dem gut parallelisierbaren „knapsack-run.input“.

Besonderheiten: Benchmark mit mittlerer Laufzeit aber äußerst feiner Granularität die noch eine Größenordnung unter der von Queens und Bitonic Sort liegt. Starke Abhängigkeit des Rekursionsbaums von den Eingabedaten.

Nur weiter aufgefächerte Rekursionsbäume wie „knap2“ bringen gute Beschleunigungen, was sich mit der Beschreibung des Cilk-Gegenstücks deckt. Bäume, in denen viel beschnitten werden kann, sind weitgehend linear, und die Feingranularität des Benchmarks verhindert im Gegensatz zum Eigenvalue Benchmark mit geometrischer Verteilung, daß Keep- und Active-Strategien erfolgreich eingesetzt werden können.

Die globale Schranke, mit der der Baum beschnitten wird, stellt prinzipiell eine unzulässige globale Variable dar, die die automatische Parallelisierung verhindert. Formal korrekt wäre nur eine Realisierung mit einem kritischen Abschnitt um die Verwendung der Schranke herum, der durch das REAPAR System nicht automatisch erkannt und

eingefügt werden kann. Solch ein Abschnitt würde außerdem bei der feinen Granularität die Leistung des Programms wesentlich verschlechtern.

In der Realität wird aber die Schranke fast nur gelesen und kaum beschrieben, und das Schreiben eines Langworts geschieht atomar. Daher kann die globale Variable für die Parallelisierung ignoriert werden. Die Ergebnisse der parallelen Läufe sind tatsächlich identisch mit denen der sequentiellen Läufe. REAPAR eignet sich also zumindest teilweise auch zur Parallelisierung von Programmen, die formal nicht in den Anwendungsbereich des Systems passen.

Da die Ablaufreihenfolge der Threads bei der Parallelisierung nur bezüglich der Vater-Kind-Beziehungen, aber nicht zwischen Kindern der gleichen Ebene oder Knoten in anderen Unterbäumen festgelegt ist, kann der Ablauf des Suchalgorithmus eine überlineare Beschleunigung erfahren. Dies tritt immer dann auf, wenn eine gute Lösung durch die andere Reihenfolge der Suche früher gefunden wird als im sequentiellen Fall und damit die restliche Suche durch eine bessere Schranke beschleunigt. Bei den Messungen ließ sich ein solches Verhalten allerdings nicht beobachten.

7.3.2 Beschleunigungen

Abbildung 7.4 führt die für alle Parallelisierungsstrategien erreichten Beschleunigungen, Systemauslastungen, Threadzahlen und System/Benutzerzeiten bei den Eingabedaten „knap2“ auf. Die Active-Strategien sind nur bis Parameter 14 aufgeführt, da die Laufzeit dann bereits bei über 4h liegt und damit eine Beschleunigung von 0,032 liefert...

Wie bei der extrem feinen Granularität des Benchmarks zu erwarten, schneidet nur die Tiefenstrategie mit geeignetem Parameter mit einer Beschleunigung von mehr als Eins ab. Sogar die Never-Strategie ist bereits deutlich langsamer als das sequentielle Neverever, da sie den Mehraufwand zum Testen der Strategiebedingung enthält.

Die folgende Tabelle zeigt die besten drei Strategien und die damit erreichten Beschleunigungen auf 4 Prozessoren sowie die Prozentzahl der Strategien, die mindestens 80% der optimalen Beschleunigung erzielen:

Problem-	1.Platz		2.Platz		3.Platz		> 80%
größe	Bes.	Strategie	Bes.	Strategie	Bes.	Strategie	
knap1	1,526	depth 2	1,381	depth 3	1,318	depth 1	6%
knap2	3,105	depth 9	3,045	depth 8	2,987	depth 12	13%

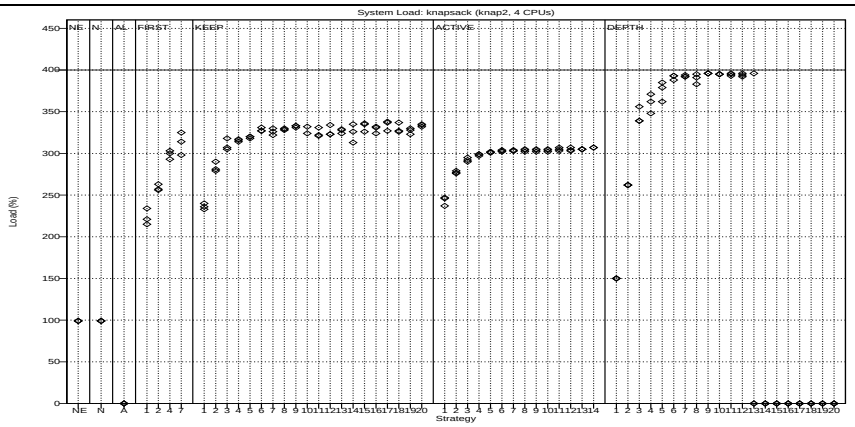
Cilk erreicht für das „knap2“-Problem eine Beschleunigung von 3,6 auf 4 Prozessoren. Dieses um 16% bessere Abschneiden erklärt sich vermutlich durch das aufwendige Laufzeitsystem von Cilk, das auf unterster Maschinenebene „Nanoschedulung“ betreibt und damit für solch äußerst feinkörnige Probleme effizienter arbeitet als REAPAR, das auf der Threadebene des Betriebssystems aufsetzt. Von den Beschleunigungen von „knap1“ berichtet Cilk nur, daß sie sehr schecht sind.

7.3.3 Strategiewahl

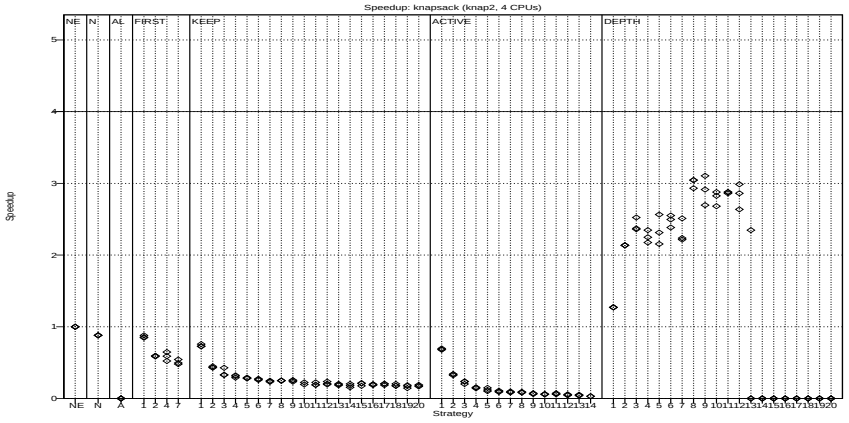
Die AP-Tiefenheuristik wählt die folgenden Strategieparameter aus:

Problem- größe	Beste reale Strategie	Gewählte Strategie	Erreichte Leistung
knap1	Depth 2	Depth 3	90%
knap2	Depth 9	Depth 2	69%

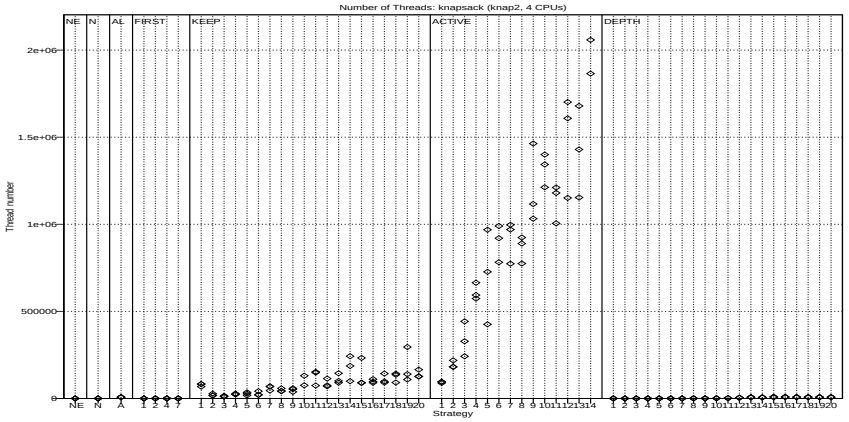
Last:



Beschleunigung:



Threadzahlen:



Zeiten:

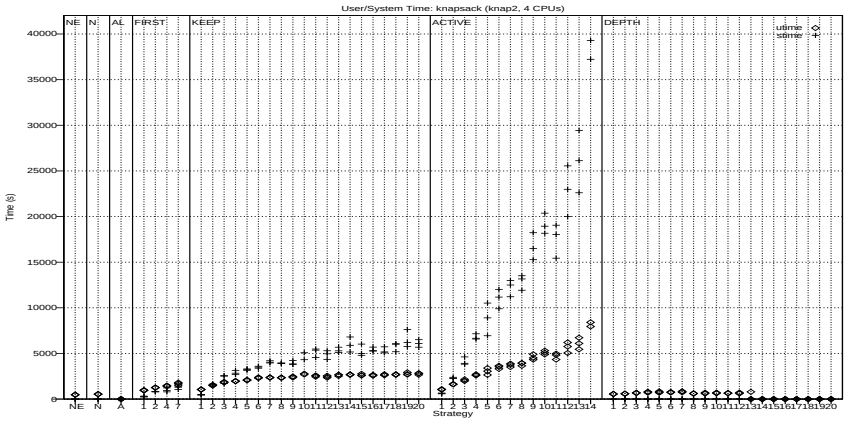


Abbildung 7.4: Knapsack — Erreichte Beschleunigungen und Systemkennwerte für alle Parallelisierungsstrategien auf vier Prozessoren für das Problem „knap2“.

Dieser Benchmark ist der einzige, bei dem die AP-Tiefenheuristik bei einem Problem mehr als 20% Leistung gegenüber dem Optimum verliert. Für „knap2“ wäre die LPS-Heuristik angemessener, die eine Tiefe von 7 empfiehlt, was immerhin 81% des Optimums entspricht. Eine Diskussion der Heuristiken findet sich im Ausblick.

7.4 Magic

7.4.1 Beschreibung

Anwendung: Kombinatorik — Finden aller möglichen $n \times n$ magischen Quadrate, d.h. Matrizen, deren horizontale, vertikale und diagonale Summe ihrer Elemente dieselbe ist.

Algorithmus: Abarbeiten einer indeterministischen virtuellen Maschine, die mögliche Lösungen rät, ihre Linearkombinationen betrachet und wirkliche Lösungen verifiziert. Die Bedingungen an die Matrix werden als lineares System umgeformt, dessen freie Variablen besetzt werden müssen, wobei die Minimierung der Kosten (Tiefe der Suche) angestrebt wird.

Parallelität: Parallelität durch gleichzeitiges Ausprobieren der verschiedenen Möglichkeiten, einen Platz zu besetzen.

Irregularität: Der implizit aufgespannte Suchbaum und damit der rekursive Ablauf ist irregulär.

Der Wurzelknoten ist typischerweise 1,5-unbalanciert, der Gesamtbaum ist (0,64; 1,00) / (0,64; 1,5) / (0,66; 3,0)-unbalanciert. Um mindestens 66% der Knoten abzudecken, reicht eine Unbalanciertheit von 1,0 aus.

Rekursion: Zwei rekursive Prozeduren, eine zur Minimierung der Kosten und eine, die die virtuelle Maschine realisiert.

Rekursionsbaum: Abbildung 7.5 zeigt einen beispielhaften Rekursionsbaum — der Baum für den 3×3 Aufruf sieht noch nicht irregulär aus, aber der Baum des 4×4 Ablaufs hat bereits 626 486 Blätter und ist zur Darstellung völlig ungeeignet.

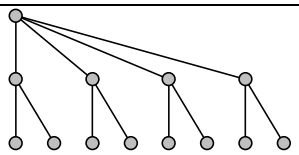


Abbildung 7.5: Magic — Rekursionsbaum für 3×3 Puzzle (siehe Anmerkung im Text).

Größe: ca. 950 LOC

Quelle: Abgeleitet aus einem Benchmark von Cilk [91].

Eingabedaten: Implizites Problem der Größe $n \times n$.

Umsetzung: Aus dem Cilk-Quellcode wurden alle Cilk-spezifischen Datentypen und Funktionen entfernt. Die zu parallelisierende `execute` Funktion, die die virtuelle Maschine rekursiv realisiert, wurde so umgeschrieben, daß sie keinen Rückgabewert, sondern einen Referenzparameter verwendet. Außerdem wurden temporäre Felder für die Zwischenspeicherung von Ergebnissen rekursiver Aufrufe eingeführt. Needresults-Annotationen stellen sicher, daß die Ergebnisse vor ihrer Verwendung vorliegen. Die zweite rekursive Prozedur zur Minimierung wurde nicht für die Parallelisierung vorbereitet, da ihr Laufzeiteinfluß gering ist. Die Parallelisierung selbst erfolgte automatisch durch das REAPAR System.

Für diesen Benchmark mußte als einziger der von REAPAR vorgegebene maximale Verzweigungsgrad von 12 auf 16 erhöht werden, da bei Eingabegröße 4 Verzweigungen von Grad 15 auftreten.

Kenngrößen: Auf einem 100MHz HyperSPARC Prozessor ergeben sich folgende Werte, wobei die Meßgenauigkeit für exaktere Aussagen über die Laufzeit bei Größe 3 nicht ausreichte — Eingabegröße 5 führt bereits zu Laufzeiten im Stundenbereich und wurde daher nicht untersucht:

Eingabe-größe	Lauf-zeit (s)	Blätter	Knoten	ms/ Blatt	Rek.Tiefe max/freq	Verz.Grad max/freq	Speicher (kb)
3	1,0	8	5	125	2/2	4/0	110
4	10,0	626 417	138 755	0,016	7/6	10/0	724

Besonderheiten: Hochdynamischer und feingranularer Benchmark, dessen Parallelisierung die oben erwähnte Einführung von temporären Feldern notwendig machte — im Originalcode wurde stattdessen das Ergebnis des rekursiven Aufrufs zu einem Zähler addiert, während das REAPAR System in seiner jetzigen Implementierung nur Prozeduren ohne Rückgabewert parallelisieren kann. Die Anpassung des Benchmarks nahm dennoch weniger als eine Stunde in Anspruch.

7.4.2 Beschleunigungen

Abbildung 7.6 gibt die für alle Parallelisierungsstrategien erreichten Beschleunigungen, Systemauslastungen, Threadzahlen und System/Benutzerzeiten bei Eingabegröße 4 an. Die Messungen für 3 sind bei einer sequentiellen Laufzeit von einer Sekunde uninteressant und die Laufzeiten für 5 überstiegen die für die Versuche verfügbare Rechenzeit bei weitem.

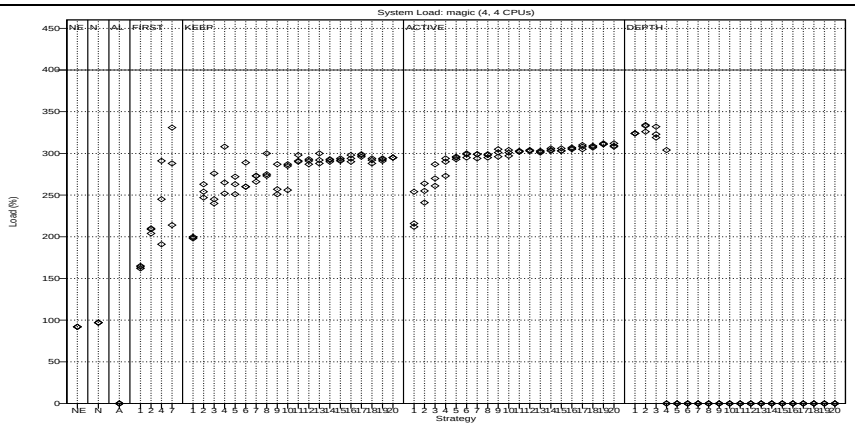
Auch hier bietet sich das von feingranularen Benchmarks inzwischen bekannte Bild: Keep und Active erzeugen zuviele Aktivitäten und haben durch die geschützte Threadzahl-Variable einen zu hohen Mehraufwand, so daß nur die Tiefenstrategie gute Beschleunigungen erreicht.

Die folgende Tabelle zeigt die besten drei Strategien und die damit erreichten Beschleunigungen auf 4 Prozessoren sowie die Prozentzahl der Strategien, die mindestens 80% der optimalen Beschleunigung erzielen:

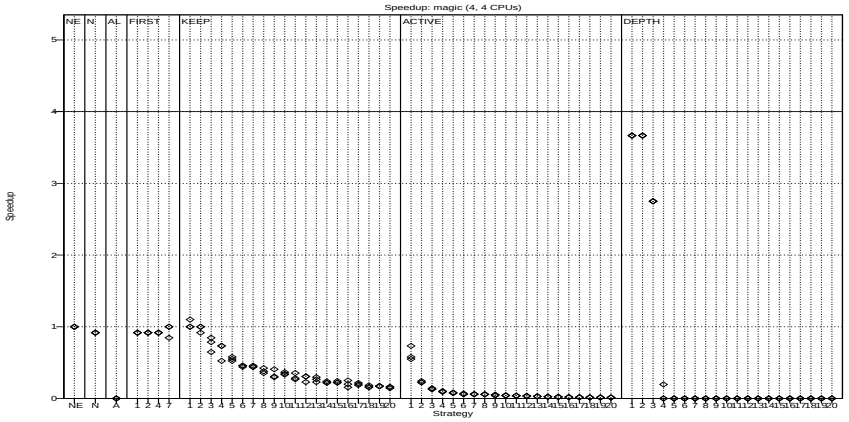
Problem-größe	1.Platz		2.Platz		3.Platz		> 80%
	Bes.	Strategie	Bes.	Strategie	Bes.	Strategie	
4	3,667	depth 1	3,667	depth 2	2,750	depth 3	4%

Cilk gibt zwar den Benchmark an, berichtet aber von keinen Beschleunigungen. REAPARs Wert von 3,7 auf 4 Prozessoren für diesen feingranularen Benchmark ist aber in jedem Falle ein gutes Ergebnis.

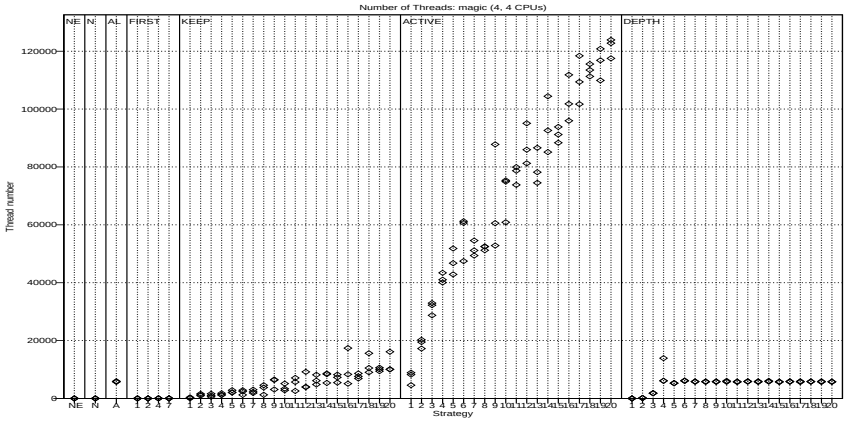
Last:



Beschleunigung:



Threadzahlen:



Zeiten:

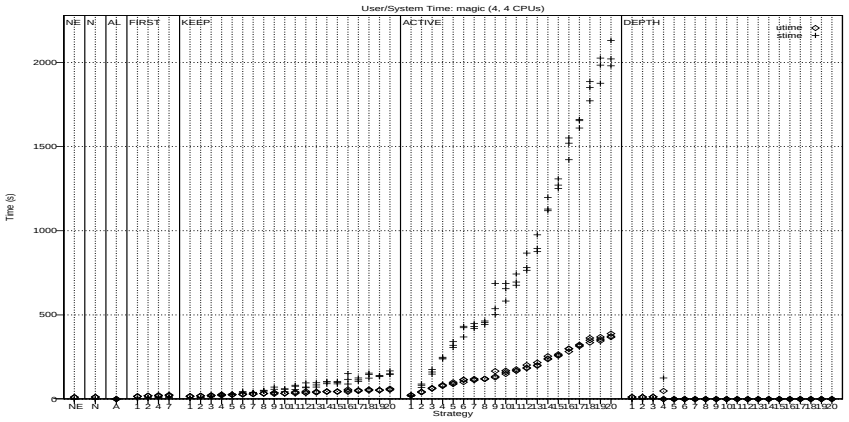


Abbildung 7.6: Magic — Erreichte Beschleunigungen und Systemkennwerte für alle Parallelisierungsstrategien, Eingabegröße 4, vier Prozessoren.

7.4.3 Strategiewahl

Die AP-Tiefenheuristik wählt für das betrachtete Problem den perfekten Strategieparameter aus, wie folgende Tabelle deutlich macht:

Problem-größe	Beste reale Strategie	Gewählte Strategie	Erreichte Leistung
4	Depth 1	Depth 1	100%

Da nur 3,5% der Strategien eine Beschleunigung von mindestens 80% des Optimums erreichen (und insgesamt sogar nur 12% der Strategien überhaupt eine Beschleunigung bewirken, die größer als Eins ist), ist die Strategiewahl hier besonders kritisch.

7.5 Heat

7.5.1 Beschreibung

Anwendung: Simulation — Verlauf der Temperaturverteilung eines Objekts durch Hitzediffusion.

Algorithmus: Die Diffusion wird simuliert durch Iterationen über eine partielle Differenzengleichung: der Hitzewert jedes Elements der Matrix berechnet sich in jedem Schritt durch Abgleich mit den Nachbarelementen. Eingabegrößen sind die Abmessungen der Matrix, die Koeffizienten für die Ausbreitungsrichtungen und die Zahl der Schritte. Die Berechnung findet mit einer Teile-und-Herrsche-Methode statt, bei der die Matrix rekursiv in Teilbereiche zerlegt wird. Die Endgröße der Teilbereiche ist ebenfalls Programmparameter („PartitionsgröSSe“).

Parallelität: Parallelität durch gleichzeitige Berechnung der beiden Teilbereiche in einem Zerteilungsschritt.

Irregularität: Der Rekursionsbaum ist im Gegensatz zu den anderen untersuchten Benchmarks ein regulärer Binärbaum, und zwar unabhängig von den Eingabedaten.

Der Wurzelknoten ist stets 1,0-unbalanciert, der Gesamtbaum ist (1,0; 1,00) / (1,0; 1,5) / (1,0; 3,0)-unbalanciert. Bereits bei einer 1-Balanciertheit werden mindestens 66% der Knoten abgedeckt. Der Rekursionsbaum ist also ein perfekter Binärbaum, wie der Algorithmus impliziert.

Rekursion: Eine zweifach rekursive Teile-und-Herrsche-Prozedur.

Rekursionsbaum: Abbildung 7.7 zeigt einen beispielhaften Rekursionsbaum.

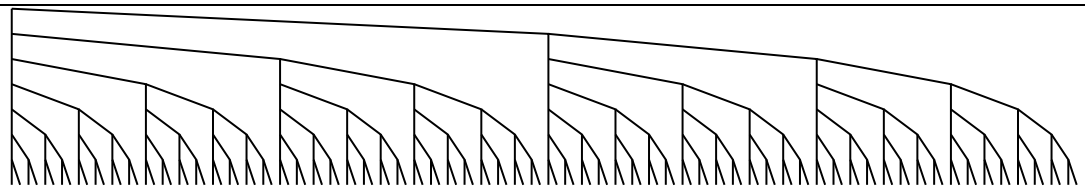


Abbildung 7.7: Heat — Rekursionsbaum für Heat 10 und Eingabesatz "heatparams".

Größe: ca. 360 LOC

Quelle: Abgeleitet aus einem Benchmark von Cilk [91].

Eingabedaten: Beispiels-Daten von Cilk, s.o.

Umsetzung: Aus dem Cilk-Quellcode wurden alle Cilk-spezifischen Datentypen und Funktionen entfernt. Die zu parallelisierende Funktion wurde so umgeschrieben, daß sie keinen Rückgabewert, sondern einen Referenzparameter verwendet. Eine Needresults-Annotation vor dem Vergleich der Ergebnisse der Teilberechnungen stellt sicher, daß diese bereits vorliegen. Zur Erhöhung der Effizienz wurde der zweite rekursive Aufruf mit Nothread annotiert. Die Parallelisierung selbst erfolgte automatisch durch das REAPAR System.

Kenngrößen: Auf einem 100MHz HyperSPARC Prozessor ergeben sich folgende Werte für den festen Eingabedatensatz „heatparams“, der auch von den Cilk-Messungen verwendet wird:

Partitions-größe	Lauf-zeit (s)	Blätter	Knoten	ms/ Blatt	Rek.Tiefe max/freq	Verz.Grad max/freq	Speicher (kb)
1	65	104 448	104 346	0,62	10/10	2/0	9 016
10	64	13 056	12 954	4,90	7/7	2/0	9 016
100	64	1 632	1 530	39,22	4/4	2/0	9 016

Besonderheiten: Dieser Benchmark ist in mehrerlei Hinsicht interessant: Zum einen ist er als prefekt reguläres Programm ein Test des REAPAR-Systems, mit solchen Problemen umzugehen. Außerdem durchläuft er wie Barnes Hut und Power mehrere Iterationen. Zum anderen fällt er als einziger Benchmark je nach Partitionsgröße in die feingranulare oder grobgranulare Kategorie, ist also auch hierbei ein guter Test für das System. Wie sich weiter unten herausstellt, erreicht die Strategiewahl von REAPAR für beide Fälle 95% des Optimums.

7.5.2 Beschleunigungen

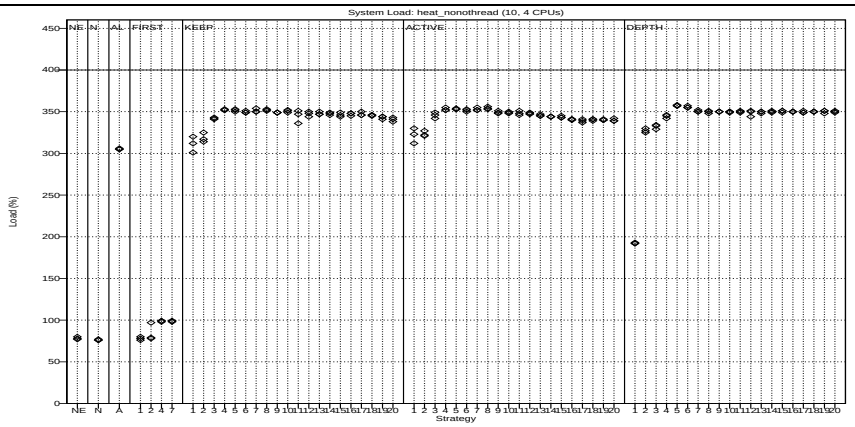
Abbildung 7.8 stellt die für alle Parallelisierungsstrategien erreichten Beschleunigungen, Systemauslastungen, Threadzahlen und System/Benutzerzeiten bei Partitionsgröße 10 dar.

Die Kurven aller Strategien ähneln sich stark — für kleine Parameterwerte wird eine perfekte Beschleunigung erreicht (aufgrund der Meßgenauigkeit sogar über 4 bei 4 Prozessoren), während große Werte den Mehraufwand erhöhen und die Beschleunigung gegen Eins drücken. Daß bei größerer Tiefe keine zusätzlichen Threads erzeugt werden, liegt daran, daß der Rekursionsbaum nur bis Tiefe 10 geht.

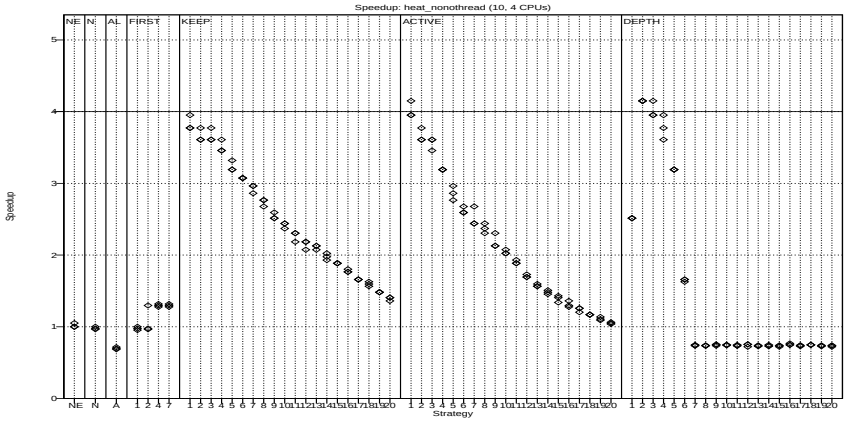
Die folgende Tabelle gibt die besten drei Strategien und die damit erreichten Beschleunigungen auf 4 Prozessoren sowie die Prozentzahl der Strategien, die mindestens 80% der optimalen Beschleunigung erzielen, an:

Partitions-größe	1.Platz		2.Platz		3.Platz		> 80%
	Bes.	Strategie	Bes.	Strategie	Bes.	Strategie	
1	3,368	Depth 2	3,368	Depth 3	3,368	Depth 4	11%
10	4,150	Active 1	4,150	Depth 2	4,150	Depth 3	16%
100	3,500	Depth 2	3,316	Active 1	3,316	Depth 3	90%

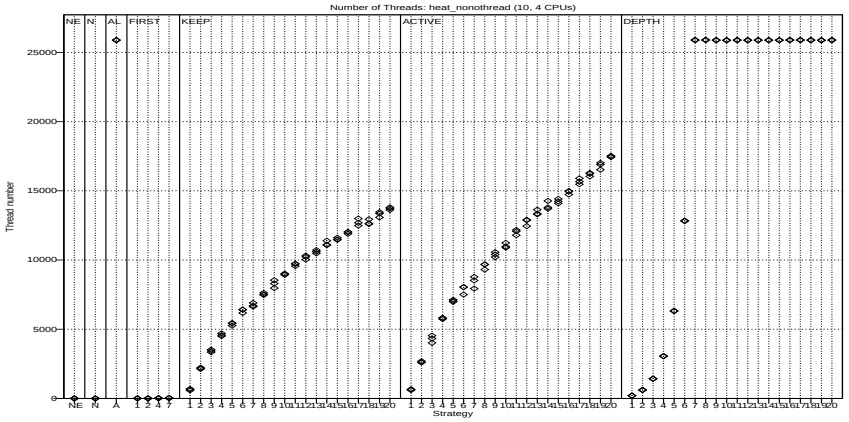
Last:



Beschleunigung:



Threadzahlen:



Zeiten:

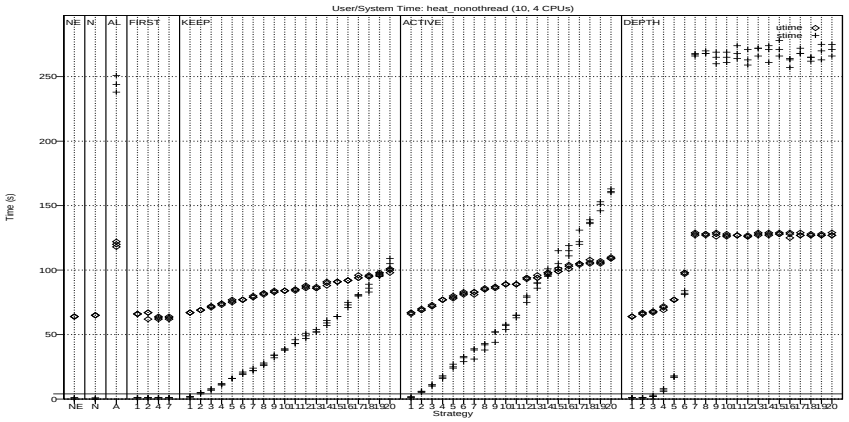


Abbildung 7.8: Heat — Erreichte Beschleunigungen und Systemkennwerte für alle Parallelisierungsstrategien, Partitionsgröße 10, vier Prozessoren.

Wie bereits im Ergebniskapitel erwähnt, übertreffen die Ergebnisse von REAPAR die Werte von Cilk für diesen Benchmark für die Partitionsgröße 10. Die feinergranularen Läufe schneiden für Nicht-Tiefenstrategien erwartungsgemäß schlechter ab, und der sehr grobgranulare 100er Lauf mit nur 16 Blättern ist für fast jede Parallelisierungsstrategie gut, wie die 90% erfolgreicher Strategien zeigen. Er schneidet etwas schlechter ab als der 10er Lauf, weil die wenigen Threads die Parallelität nicht perfekt ausnützen können.

7.5.3 Strategiewahl

Die Strategie für das feingranulare Heat mit Partitionsgröße 1 wurde durch die AP-Tiefenheuristik ermittelt, die für den perfekten Binärbaum des Problems dasselbe Ergebnis wie die LPS-Heuristik liefert. Die Probleme der anderen beiden Größen werden simuliert, was insgesamt zu folgender Strategiewahl führte:

Partitions-größe	Beste reale Strategie	Gewählte Strategie	Erreichte Leistung
1	Depth 2	Depth 3	95%
10	Active 1	(1) Depth 4	95%
		(2) Active 2	91%
		(3) Keep 3	91%
100	Depth 2	(1) Depth 3 [x]	95%
		(2) Keep 3	86%
		(3) Active 1 [x]	95%

Die mit [x] gekennzeichneten Strategien bei Partitionsgröße 100 werden von der Auswahlheuristik nicht empfohlen, weil für sie die Zahl der Knoten im Unterbaum eines Threads nicht kleiner als 1/(*Zahl der Prozessoren*) wird — der Rekursionsbaum bei dieser Größe hat nur eine Tiefe von 4 und damit 16 Blätter und 15 Knoten.

7.6 Fazit

Die Anwendung des fertigen Systems auf bisher nicht betrachtete Benchmarks erzielt sehr gute Ergebnisse bei der Beschleunigung, die den Vergleichswerten aus der Literatur nahekommen oder sie sogar übertreffen. Auch die Strategiewahl wählt solche Parameter, die dem Optimum sehr nahekommen oder es erreichen (von Einbußen bei Magic abgesehen).

Die Validierungsmenge schneidet dabei nicht schlechter ab als die für den Systemaufbau verwendeten ursprünglichen fünf Benchmarks, so daß von keiner Optimierung des Systems auf die Ur-Benchmarks hin die Rede sein kann.

Damit ist der Beweis erbracht, daß das System nicht nur auf den zur Konstruktion verwendeten Benchmarks funktioniert, sondern auch für völlig neue Programme sehr gute Leistungen zeigt.

Kapitel 8

Zusammenfassung und Ausblick

Dieses Kapitel faßt die Kernpunkte und Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf weitere Forschung im Umfeld der Parallelisierung irregulärer rekursiver Programme.

8.1 Ergebnisse der vorliegenden Arbeit

Im Rahmen der Arbeit habe ich die anfangs in Kapitel 4 aufgestellten Thesen bewiesen:

- Ich habe effiziente Parallelisierungsstrategien speziell für irreguläre rekursive Programme aufgestellt, die in der Praxis bei bis zu acht Prozessoren gute bis maximale Beschleunigungen erreichen.
- Das im Rahmen der Arbeit entwickelte REAPAR-System parallelisiert Programme automatisch durch Quellcodetransformation für diese Strategien.
- Relevante Laufzeitdaten des Programms werden durch eine automatische Instrumentierung des Quellcodes aufgezeichnet.
- Das REAPAR-System analysiert automatisch die aufgezeichneten Daten und wählt daraufhin eine für das Problem geeignete Parallelisierungsstrategie durch eine Kombination von Threadablauf-Simulation und Heuristiken.
- Die Strategiewahl selektiert fast immer die Strategie, die in der Realität am besten abschneidet, oder liegt sehr nahe am Optimum.
- Durch eine Kontrollmenge von vier zusätzlichen Benchmarks wurde die Leistung des Systems für bisher unbekannte Programme nachgewiesen.
- Eine Beschleunigung von bis zu 33 (sic!) auf 30 Prozessoren eines Testrechners belegt die Skalierbarkeit der Verfahren für hinreichend grobgranulare Programme.
- Im Literaturvergleich erreicht das REAPAR-System die Leistung vergleichbarer Systeme, die auf manuellen Parallelisierungen basieren, oder übertrifft sie.
- Im Gegensatz zu einer Handparallelisierung benötigt das System für die automatische Parallelisierung und Strategiewahl nur einen Bruchteil der Zeit, erzielt ähnliche Ergebnisse und setzt keine weitergehenden Kenntnisse der Parallelrechnerprogrammierung voraus.

8.2 Weitere Richtungen der Forschung

Wie im Entwurfskapitel erwähnt, konzentriert sich REAPAR auf die automatische Parallelisierung und führt selbst z.B. keine **Datenabhängigkeitsanalysen** durch. Die Schnittstelle zu Systemen, die solche Analysen vornehmen können, wird in Form von Annotationen des Quellcodes angeboten. Durch den Einsatz solcher Systeme könnte die Arbeit des Benutzers noch weiter reduziert werden, indem etwa automatisch Prozeduren mit sequentialisierenden Datenabhängigkeiten von der Parallelisierung ausgeschlossen werden oder das System Stellen zur vorzeitigen Zusammenführung der erzeugten Threads automatisch erkennt und für REAPAR annotiert.

Die Methoden der vorliegenden Arbeit können ohne Änderungen auch für andere imperative Programmiersprachen verwendet werden. **Objektorientierte Sprachen** stellen hingegen eine gewisse Herausforderung dar, weil statisch nicht bekannt ist, welche Methode welchen Objekts wo aufgerufen wird. Aus genau diesem Grunde verbietet REAPAR Funktionszeiger. Allerdings greifen meine Verfahren zur Instrumentierung von Programmen genauso im OO-Fall, z.B. können Methoden um einen Tiefenparameter ergänzt werden, und Aufrufstatistiken sind ebenfalls möglich. Dadurch läßt sich zur Laufzeit ein OO-Analogon des Rekursionsbaums aufbauen, das wie dieser über Informationen der Aufrufe und Abfolgen verfügt. In diesem OO-Rekursionsbaum können dann genau wie in dieser Arbeit beschrieben Threadablauf-Simulationen durchgeführt werden, die Hinweise auf geeignete Parallelisierungen geben können. Die Parallelisierungsstrategien gelten für Methodenaufrufe genauso wie für Prozeduraufrufe, so daß eine Anpassung von REAPAR für OO-Sprachen möglich erscheint.

Eine Erweiterung des REAPAR-Ansatzes wäre eine **dynamische Anpassung der Parallelisierungsstrategie** zur Laufzeit. Das parallele Programm könnte periodisch die Effizienz der aktuellen Parallelisierungsstrategie messen und den Strategieparameter oder sogar die Strategie selbst wechseln, wenn die Leistung der Maschine nicht voll ausgenützt wird. Wie bereits in Abschnitt 4.3.2.2 beschrieben, erfordert ein solches Vorgehen einen großen Aufwand bei der Infrastruktur, so daß ich in der vorliegenden Arbeit davon Abstand nahm. Die mit statischer Strategiewahl erzielten Beschleunigungen geben dem recht, aber einige von ihnen ließen sich vielleicht mit dynamischer Strategiewahl weiter verbessern. Eine Strategiewahl zur Laufzeit könnte zudem die initialen Programmläufe zur Datensammlung überflüssig machen, wodurch das System sofort produktiv einsetzbar wäre und nicht erst nach einem Testlauf. Außerdem könnte Programmen besser Rechnung getragen werden, deren Rekursionsbaum sich über mehrere Iterationen hinweg verändert und eventuell anfangs andere Strategien bevorzugt als gegen Ende der Berechnung.

Falls der Programmierer ein tiefgehendes Verständnis seines Codes hat, wobei erfahrungsgemäß die Intuition über das Programmverhalten keine große Korrelation mit der Realität haben muß, sind auch **weitergehende Annotationen** denkbar: Genaue Informationen über die zu erwartende Größe des Unter-Rekursionsbaums einer Prozedur in Abhängigkeit von Laufzeitwerten wäre evtl. ebenso wertvoll für Optimierungen wie eine Formel für den Berechnungsaufwand des Gesamtproblems als Funktion der Problemgröße oder eine a priori Einstufung des Problems als grob- oder feingranular. Mit solchen Informationen könnte auch die erwähnte dynamische Strategiewahl zusätzlich verbessert werden.

Programme mit „zerfaserten“ Rekursionsbäumen wie der Knapsack-Benchmark würden bei der Strategiewahl durch die LPS-**Heuristik** besser abschneiden als mit der optimistischeren AS-Heuristik. Wenn REAPAR durch Analyse des Laufzeitprofils auf die Baumform hin entscheiden könnte, welche Heuristik angemessener einzusetzen ist, ließe sich die Leistung

des Systems für solche Programme weiter steigern.

Für extrem **feingranulare** Programme erzielt REAPAR auf Rechnern mit mehr als 8 Prozessoren offenbar keine guten Leistungen mehr, weil selbst bei der Tiefenstrategie der Mehraufwand durch Instrumentierung und Abfrage des Threaderzeugungs-Prädikats zu groß wird und die Laufzeit eines Einzelthreads zu gering ist. Systeme wie Cilk verwenden weitgehende Eingriffe in das Betriebssystem (Nanoscheduling), um auch für solche Fälle eine gute Parallelisierung zu erzielen. Von solchen Techniken könnte auch REAPAR profitieren, allerdings auf Kosten des portablen, minimalistischen Ansatzes und mit hohem Realisierungsaufwand für den Nanoscheduler.

Rechner mit **verteiltem Speicher** stellen eine zusätzliche Herausforderung dar, da hier eine gute Verteilung von parallelen Aktivitäten und die Lokalität der benötigten Daten unbedingte Voraussetzungen für eine effiziente Parallelisierung sind. REAPAR muß in seiner jetzigen Form diese Aspekte gar nicht berücksichtigen, da sie durch Maschinen mit gemeinsamem Speicher und ihre Betriebssysteme bereits behandelt werden. Sowohl Lastverteilung als auch Datenlokalität sind Gegenstand vieler anderer Arbeiten. Interessant wäre die Integration von REAPAR mit einem System, das diese Aspekte berücksichtigt.

Zusammenfassend läßt sich sagen, daß sich einige interessante Richtungen für weitere Forschung und Ergänzungen des REAPAR-Systems anbieten. Dennoch übertrifft das System bereits in seiner jetzigen Form die Entwurfsanforderungen, was die Qualität des gewählten Ansatzes unterstreicht.

Anhang A

A.1 Hilfsprogramme und Werkzeuge

Im Rahmen der Arbeit wurde eine Vielzahl von Zusatzprogrammen entwickelt, die bei der Durchführung und Auswertung von Meßreihen, der Visualisierung und weiteren Aufgaben eine große Erleichterung waren. Dieser Abschnitt führt ohne Anspruch auf Vollständigkeit einige von ihnen auf, um einen Eindruck des Gesamtsystems zu vermitteln.

`komplette_messreihe` P C S**:** Instrumentiert und parallelisiert das Programm P und führt es auf C CPUs mit der Eingabegröße S für alle möglichen Parallelisierungsstrategien aus. Aus den Ergebnissen werden die Zahl der erzeugten Threads, die Laufzeit, die Systemzeiten und die Maschinenauslastung herausgefiltert und in eine Auswertungsdatei geschrieben.

`do_plots` $A_1 \ldots A_n$**:** Wertet die gegebenen Auswertungsdateien aus und erzeugt mit dem Hilfsskript `make_plot_from_messreihe.awk` eine Datei, aus der per `gnuplot` automatisch die Graphen der Auslastung, Beschleunigung, Systemzeiten und Threadzahlen erzeugt werden, wie sie in Kapitel 6 zu sehen sind.

`top3_to_tex` $A_1 \ldots A_n$**:** Sucht aus den gegebenen Auswertungsdateien die jeweils besten drei Parallelisierungsstrategien mit ihren Beschleunigungen heraus, bevorzugt dabei bei Gleichstand solche Strategien, die einen kleinen Wert für den Strategieparameter haben. Die Ergebnisse finden sich z.B. in Tabelle 6.2.

`make_ranked_data_list:` Sortiert die Ergebnisse einer Meßreihe nach erreichter Beschleunigung und ordnet sie in Plätze ein. Strategien mit gleicher Beschleunigung erhalten den selben Platz, d.h. es kann mehrere erste Plätze geben. Außerdem für jede Strategie berechnet, wieviel Prozent der optimalen Beschleunigung sie erreicht, so daß sich danach z.B. für die Bewertung der automatischen Strategiewahl das reale Abschneiden einer Strategie sehr schnell beurteilen läßt.

`percentage_greater_x` M P**:** Stellt anhand der sortierten Meßreihe M fest, wieviel Prozent der Strategien mindestens $P\%$ der optimalen Beschleunigung erreichten.

`compare_top_3_strategies` $M_{1\ldots n}$**:** Ermittelt in allen gegebenen sortierten Meßreihen verschiedener Probleme, d.h. Läufe eines Benchmarks mit verschiedenen Eingabedaten, wie gut die für ein bestimmtes Problem beste Strategie für alle anderen Probleme abschneidet. Dadurch läßt sich feststellen, wie empfindlich eine einmal gewählte Strategie gegenüber Änderungen des Problems ist. Die Ergebnisse finden sich in den vom Werkzeug automatisch erzeugten Tabellen 6.6 bis 6.10.

`compare_real_threads_with_simulation` E: Vergleicht aufgrund der Eingabedefinitionsdatei E statistische Kenngrößen der Threadzahlen, die in Versuchen erzielt wurden, mit denen der simulierten Threadzahlen und führt dazu eine Simulation der entsprechenden Programme basierend auf ihrem Rekursionsbaum durch. Dieses Skript wurde verwendet, um die Realitätsnähe der Simulation bezüglich der Threadzahl zu beurteilen, zusammen mit `collect_threadinfo_from_measurement_rawdata_histogram`, das die Zahl der real erzeugten Threads grafisch in Histogrammen aufbereitet — eine Diskussion findet sich in Abschnitt A.3.

`tree_graph_output`: Erzeugt aus dem gegebenen textuellen Rekursionsbaum eines Programms, wie er in Abschnitt 5.4.5 gezeigt wurde, eine grafische Darstellung des Baums. Dieses C-Programm wurde bereits im erwähnten Abschnitt beschrieben.

`tree_from_profile`: Errechnet aus dem Laufzeitprofil eines Programms den zugehörigen abgeleiteten Rekursionsbaum (WCT), wie er z.B. in Abbildung 5.8 dargestellt ist.

`analyze_balance` B: C-Programm, das den Rekursionsbaum aufgrund der Balanciertheitskriterien aus Abschnitt 2.4 analysiert und die Balanciertheit des Wurzelknotens und die (x,B)-Balanciertheit des Baums ausgibt. Basis für die Balanciertheitsaussagen der Benchmarkbeschreibungen.

A.2 Versuche zum Maschinenlernen

Wie in Abschnitt 4.3.2.2 besprochen, bietet sich das Maschinenlernen als Methode zur Strategiewahl an, weil sich automatisch große Datenmengen als Grundlage des Lernens gewinnen lassen. Idealerweise würde das System mit Beispielsprogrammen und ihren Beschleunigungen für alle möglichen Strategien angelernt und könnte dann als *black box* die Beschleunigungen neuer Programme vorhersagen. Damit wäre eine Wahl der als schnellsten vorhergesagten Strategie möglich.

Vor diesem Hintergrund wurden in den Anfängen der Realisierung von REAPAR Maschinenlernverfahren untersucht — eine Einführung in neuere Maschinenlerntechniken bietet der Report von D.Michie et al. [68]. Zum Einsatz kam bei den REAPAR Experimenten das System MARS [37] in der Version von Delve [80]. MARS (Multivariate Adaptive Regression Splines) versucht, mehrdimensionale Punktemengen durch Splines bei minimalem Fehler zu approximieren, d.h. das Lernergebnis ist eine Spline-Funktion in den Eingabeparametern, die den Ausgabeparameter annähert.

Eingabe des Systems waren die Meßwerte aller fünf ursprünglichen Benchmarks, wie sie in der Datensatzdefinition A.1 abgebildet sind. Wie man sieht, wurden hier auch die später verworfenen kombinierten Strategien getestet. Diese Rohdaten sollten zur Klassifizierung des Problems und zur Vorhersage seiner Beschleunigung dienen.

Die aus den fünf ursprünglichen handparallelisierten Benchmarks gewonnenen Meßwerte für 4, 6 und 8 CPUs wurden zufällig in eine Lern- und eine Kontrollmenge aufgeteilt. Pro Benchmark wurden die Ergebnisse von 2 000 bis 7 200 Messungen betrachtet, insgesamt 19 400 Datensätze. Als Variante wurden die Datensätze nach CPU-Zahl und nach Strategieklasse (Tiefen, Allgemein, Kombiniert) getrennt.

Danach wurde MARS auf der Lernmenge gestartet, um eine Annäherung der Daten zur Vorhersage der Beschleunigungen zu bilden. Die Qualität der Vorhersage wurde dann durch statistische Analyse der Abweichungen zur Kontrollmenge beurteilt.

```
Title: Recursive Irregular
Origin: natural
Usage: assessment
Order: uninformative
Attributes:
 1 BENCHMARK      c BARNES EIGENVALUE FRACTAL POWER QUEENS  # For information only
 2 STRATEGY_NAME  c ALWAYS NEVER KEEP_N FIRST_N DEPTH_ONLY COMBINED # Name of the
                                                #   strategy (general/depth/both)
 3 USE_GENERAL    c 0 1                         # Use general strategy?
 4 USE_DEPTH      c 0 1                         # Use depth strategy?
 5 GENERAL_PARAM  c 0..Inf                      # Parameter N for Strategy
 6 DEPTH_PARAM    c 0..Inf                      # Maximum thread generation depth
 7 NUM_CPU        c 1..Inf                      # Number of CPUs used
 8 INPUT_SIZE     c 0..Inf                      # Size of input
 9 DEPTH_REACHED  u 0..Inf                      # Recursion depth reached
10 LEAFS          u 0..Inf                      # Number of Leafs in recurs. tree
11 WALLCLOCK      u 0..Inf                      # Program runtime (seconds)
12 LEAF_TIME      u [0,Inf)                     # Time for one leaf
13 SPEEDUP        u [0,Inf)                     # Speedup obtained
14 THREADS        u 1..Inf                      # Number of threads created
15 USER_SEC       u [0..Inf)                    # Seconds spent in user mode
16 SYS_SEC        u [0..Inf)                    # Seconds spent in system mode
17 PERC_USAGE     u 0..Inf                      # Percentage of machine usage
18 SEQTIME        u [0..Inf)                    # Sequential runtime
```

Abbildung A.1: Definition der Eingabedaten für MARS

Außerdem wurde die relative Reihenfolge der geschätzten Beschleunigungen mit ihrer tatsächlichen Reihenfolge verglichen — zur Wahl einer guten Strategie reicht ja die relative Einordnung der möglichen Strategien völlig aus, auch wenn die absolut geschätzte Beschleunigung nicht perfekt genau ist. Eine Alternative wäre gewesen, keine Beschleunigungen vorherzusagen sondern nur das paarweise relative Abschneiden der Strategieen.

Die Auswertung der vorhergesagten Reihenfolgen ergab aber, daß z.B. für die kombinierten Strategien die Reihenfolge der besten realen Strategien genau umgekehrt zu den besten geschätzten Strategien ist und die Breite der Schätzungen für einzelne Benchmarks viel enger als die reale Schwankung der Beschleunigungen ausfällt. Außerdem besteht die Vermutung, daß für manche Benchmarks, die sich durch besondere Eingabegrößen aus den Daten hervorheben, die Beschleunigungen einfach auswendig gelernt wurden. Für Eingabegrößen ungleich der Lern-Eingabegrößen waren die Schätzungen sehr inkonsistent. Auch die absoluten Beschleunigungsschätzungen lieferten keine überzeugenden Ergebnisse.

Das ursprüngliche Vorhaben, das Maschinenlernen als *black box* zur Strategiewahl verwenden, war also nicht erfolgreich. Für einen realen Einsatz hätte man zudem noch Verfahren entwickeln müssen, die nur aufgrund der Kenngrößen eines Programms bereits seine Beschleunigung vorhersagen können, da in der Praxis nicht die Ergebnisse sämtlicher Strategien bekannt sind. Dazu wären weitergehende Daten nötig gewesen, da z.B. die Größe der Eingabe eines Programms extrem problemspezifische Bedeutung hat (100 Zahlen zu sortieren ist ein anderer Aufwand, als 100 Sterne einer Galaxie zu simulieren). Eine Möglichkeit

wäre etwa, das binäre Kriterium „Granularität“ aus der Blatt-Laufzeit abzuleiten und danach vorzuklassifizieren. Nötig wären vermutlich weitergehende, bereits aufbereitete Daten gewesen, die z.B. die Form des Rekursionsbaums zusammenfassen. Die Erfahrungen mit der letztendlich realisierten Strategiewahl legen nahe, daß die Baumform und die Granularität einen entscheidenden Einfluß auf die gute Strategiewahl haben. Dementsprechend wären auch Statistiken des Rekursionsprofils wie Schwankungen der Blattzahl, das Verhältnis von Blättern zu Knoten oder Rekursionstiefen, die 50% bzw. 90% der Blätter bzw. Knoten eines Baums abdecken, als Kennzahlen verwendbar.

Bei geeigneter Wahl solcher zusätzlichen Datenquellen, die sich fürs Lernen auf Skalare abbilden lassen, wäre das Maschinenlernen evtl. eine interessante Alternative, da die dazu nötigen Datenmengen automatisch zur Verfügung gestellt werden könnten. Die folgende Entwicklung von REAPAR in Richtung der kombinierten Simulation und Heuristiken erwies sich aber als so erfolgreich, daß der Maschinenlernansatz nicht weiter verfolgt wurde.

A.3 Versuche zur Threadzahl-Vorhersage

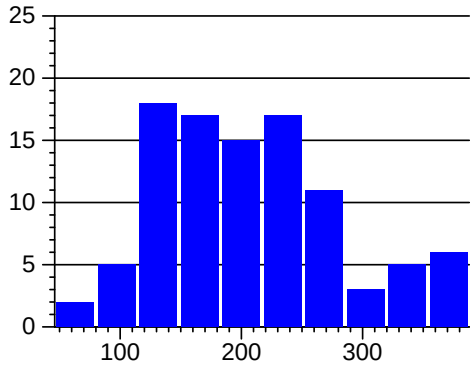
In einer frühen Phase der Arbeit wurde untersucht, wieviele Threads von welcher Parallelisierungsstrategie an welchen Stellen des Rekursionsbaums erzeugt werden, um ein besseres Verständnis der Threaderzeugung zu erhalten. Viele Aspekte der Threaderzeugung sind indeterministisch: Bei der Tiefenstrategie ist zwar die Zahl der Threads durch Rekursionsbaum und Maximaltiefe festgelegt, aber die Reihenfolge ihrer Erzeugung hängt auf mehreren Prozessoren vom Scheduler des Betriebssystems ab. Bei allgemeinen Strategien (von den trivalen Never und Always abgesehen) ist zusätzlich unbekannt, an welcher Stelle des Programmablaufs Threads ihre Arbeit beenden und dadurch das Threaderzeugungsprädikat für einen rekursiven Aufruf an anderer Stelle aktiv werden lassen.

Zur Bestimmung der Threaderzeugung bei den allgemeinen Strategien wurden daher zusätzliche Felder im Programm eingeführt, die bei Erzeugung und Beendigung eines Threads seine anfängliche Rekursionstiefe, seine maximale erreichte Tiefe, die Zahl der von ihm bearbeiteten Knoten und (implizit) die Reihenfolge seiner Erzeugung aufzeichneten. Da die erreichten Threadzahlen bei längeren Benchmarkläufen zum Teil stark schwanken, wurden auch pro Eingabegröße, Benchmark und Strategie je 100 Läufe durchgeführt, um die Verteilung der Threadzahlen zu erfassen. Zwei typische Histogramme solcher Verteilungen, die die Schwankungsbandbreite der Threadzahlen verdeutlichen, finden sich in Abbildung A.2. Man sieht, daß im gleichmäßigen uniformen Rekursionsbaum weniger Threads erzeugt werden, da im Gegensatz zum unbalancierten geometrischen Baum die einzelnen Threads länger laufen können, bis ihre Unterbaum abgearbeitet ist.

Außerdem wurde mittels der in Abschnitt A.1 erwähnten Werkzeuge weitgehende Statistiken der Rekursionsbäume und zugehörigen Threadverteilungen erstellt. Vor diesem Hintergrund wurde dann versucht, die Zahl der in der Simulation „erzeugten“ Threads in Relation zu den gemessenen Threadzahlen zu setzen. Besonders für große Benchmarkläufe wurden aber zuwenige Threads simuliert.

Weitere Überlegungen ergaben jedoch, daß es weniger wichtig ist, die genaue Threadzahl zu simulieren — entscheidend ist vielmehr, die relative Reihenfolge der Laufzeiten in der Simulation möglichst realitätsnahe zu beschreiben. Dieser Ansatz brachte dann auch den gewünschten Erfolg, wie die vorliegende Arbeit zeigt.

Eigenvalue uniform 4096



Eigenvalue geometrisch 4096

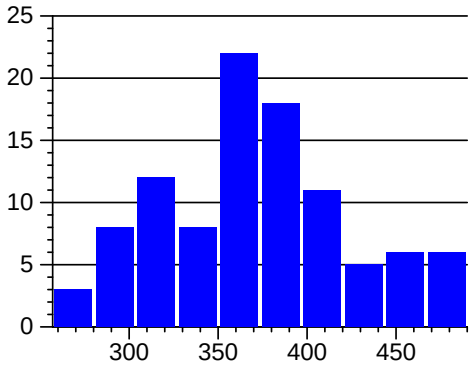
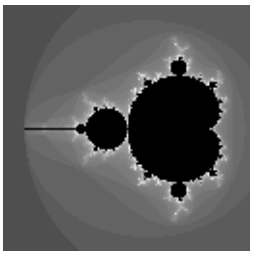


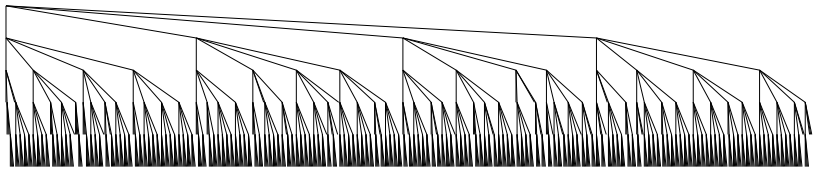
Abbildung A.2: Verteilung der erreichten Threadzahlen bei Eigenvalue uniform und geometrisch über 100 Läufe, jeweils mit der Strategie Active 5.

A.4 Korrespondenz von Rekursionsbaum und Bildausschnitt bei Fractal

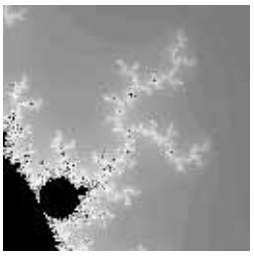
Der Fractal-Benchmark zeigt auf einzigartige Art und Weise den Zusammenhang von Rekursionsbaum und Problem auf: Das Ergebnis des Programmlaufs ist ein Bild, dem man die Rekursionstiefe der zugehörigen Berechnung leicht ansehen kann. Bereiche, die großflächig dieselbe Farbe/Graustufe haben, werden durch die Heuristik schnell ausgefüllt und bedeuten geringe Rekursionstiefen. Bereiche mit vielen Farbänderungen verlangen nach höheren Rekursionstiefen. Abbildungen A.3 stellt die fünf für die Benchmarkläufe verwendeten Bildausschnitte den zugehörigen Rekursionsbäumen gegebüber (Bildauflösung 128×128, Auflösung der Rekursionsbäume 64× 64). Alle Rekursionsbäume sind maßstäblich, d.h. der Baum von Problem 3 enthält tatsächlich nur etwa die Hälfte der Arbeit von Problem 4.

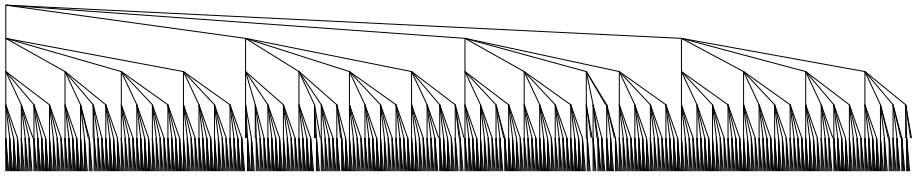
Bildausschnitt 1: Ur-Fraktal, symmetrisch, Unvollständigkeit der Rekursionsteilbäume auf dem gesamten Bild etwa gleichmäßig verteilt. Startkoordinaten (-2,25 , -1,5) Ausschnittgröße 3,0



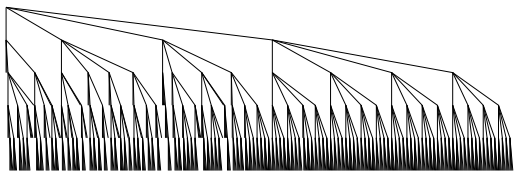


Bildausschnitt 2: Ungleichmäßig verteilte Arbeit, einige Rekursionsteilbäume sind vollständiger als andere. Startkoordinaten (-0,0585 , -0,8850) Ausschnittgröße 0,1200

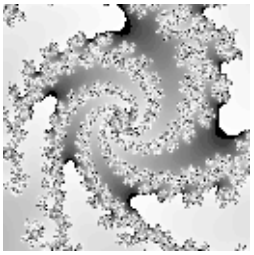


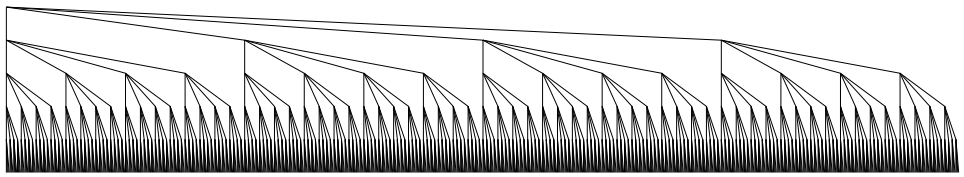


Bildausschnitt 3: Nur Arbeit in unterer rechter Ecke, d.h. vollständiger Rekursionsunterbaum für den entsprechenden Quadranten und extrem spärliche Unterbäume für den Rest des Bildes. Startkoordinaten (-0,013095 , -0,810949) Ausschnittgröße 0,000038



Bildausschnitt 4: Asymmetrisch, aber überall viel Arbeit, dementsprechend vollständige Rekursionsunterbäume überall. Startkoordinaten (-0,53802 , -0,52812) Ausschnittgröße 0,00480





Bildausschnitt 5: Wenig Arbeit in unterer rechter Ecke mit entsprechender Unterbeledung des rechten Teilbaums. Startkoordinaten (-0,764412 , -0,101004) Ausschnittgröße 0,000960

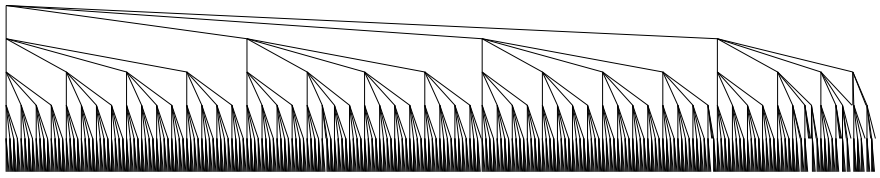


Abbildung A.3: Gegenüberstellung von Bildausschnitten und entsprechenden Rekursionsbäumen für die fünf Fractal-Probleme. Die rechte untere Ecke entspricht dem rechten Viertel des Rekursionsbaums, alle Bäume sind im selben Maßstab abgebildet.

Literaturverzeichnis

[1] J.M. Anderson, S.P. Amarasinghe, and M.S. Lam. Data and computation transformations for multiprocessors. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, ACM SIGPLAN Notices 30(11), pages 166–178, August 1995.

[2] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison Wesley Longman, Reading, MS, USA, second edition, 1998.

[3] Tom H. Axford. *The Divide-and-Conquer Paradigm as a Basis for Parallel Language Design*, volume "Advances in Parallel Algorithms", chapter 2. Blackwell, 1992.

[4] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer, Boston, USA, 1993.

[5] Joshua Edward Barnes and Piet Hut. A hierarchical $O(N \log N)$ force calculation algorithm. *Nature*, 324:446, 1986.

[6] Kenneth E. Batcher. Bitonic sorting. Technical Report GER-11759, Goodyear Aerospace Report, 1964.

[7] Aart J.C. Bik and Dennis B. Gannon. Automatically exploiting implicit parallelism in Java. Technical report, Computer Science Dept., Indiana University, Bloomington, IN, USA, 1997.

[8] Aart J.C. Bik, Juan E. Villacis, and Dennis B. Gannon. `javar` *Manual (version 1.3beta)*. Computer Science Department, Indiana University, Bloomington, IN, USA, 1997.

[9] Gianfranco Bilardi and Alexandru Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines. Technical Report TR86-769, Department of Computer Science, Cornell University, Ithaca, NY, USA, August 1986.

[10] Tom Blank. The MasPar MP-1 architecture. In *Proceedings of the COMPCON Spring 1990*, pages 20–24, San Francisco, CA, USA, February 26 – March 2, 1990. IEEE Computer Society.

[11] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, March 1996.

[12] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995.

[13] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *USENIX 1997 Annual Technical Symposium*, Anaheim, CA, USA, January 6–10, 1997. USENIX.

[14] Tore A. Bratvold. Determining useful parallelism in higher order functions. In Herbert Kuchen and Rita Loogen, editors, *4th International Workshop on the Parallel Implementation of Functional Languages*, Aachen, Germany, September 1992. Heriot-Watt University, UK.

[15] California Institute of Technology, Pasadena, CA, USA, `http://www.cs.caltech.edu/~threads/multithreaded_c/reference.html`. *Multithreaded C Reference Manual*, 1st edition, February 1997.

[16] Martin C. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Princeton University Department of Computer Science, June 1996.

[17] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. In *PPOP 1995*, Princeton University, USA, July 1995. ACM SIGPLAN.

[18] Soumen Chakrabarti, Abhiram Ranade, and Katherine Yelick. Randomized load balancing for tree-structured computation. In *Scalable High Performance Computing Conference*, pages 666–673, Knoxville, USA, 1994. IEEE.

[19] Soumen Chakrabarti and Katherine Yelick. Implementing an irregular application on a distributed memory multiprocessor. In *Principles and Practice of Parallel Programming (PPOPP'93)*, pages 169–178, San Diego, CA, USA, May 1993.

[20] K.M. Chandy and C. Kesselman. CC++: A declarative, concurrent, object oriented programming notation. Technical Report CS-92-01, California Institute of Technology, Pasadena, CA, USA, 1992.

[21] Terry W. Clark, Reinhard v. Hanxleden, J. Andrew McCammon, and L. Ridgway Scott. Parallelization strategies for a molecular dynamics program. In *Technology Focus Conference, Timberline Lodge 1992*, Houston, TX, USA, April 5–7, 1992. Intel University Partners Program, Supercomputer Systems Division.

[22] Terry W. Clark, Reinhard v. Hanxleden, J. Andrew McCammon, and L. Ridgway Scott. Parallelization using spatial decomposition for molecular dynamics. Technical Report CRPC-TR93356-S, Rice University, Center for Research on Parallel Computation, Houston, TX, USA, November 1993.

[23] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT electrical engineering and computer science series. MIT Press, Cambridge, Massachusetts, USA, 1990.

[24] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven S. Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Supercomputing 1993 Proceedings*, Berkeley, CA, USA, 1993.

[25] David E. Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Schauser, Eunice Santos, Ramesh Sumbramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Principles and Practice of Parallel Programming*, pages 1–12, San Diego, CA, USA, May 1993.

[26] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel programming using skeleton functions. In *PARLE*, 1993.

[27] Raja Das, Joel Saltz, and Reinhard v. Hanxleden. Slicing analysis and indirect accesses to distributed arrays. Technical Report CRPC-TR93319-S, Rice University, Center for Research on Parallel Computation, Houston, TX, USA, June 1993.

[28] Jack Dongarra, Hans-Werner Meuer, and Erich Strohmaier. Top 500 Supercomputer sites. `http://parallel.rz.uni-mannheim.de/top500/top500.list.html`, July 1997.

[29] Hans Eberle. Architektur moderner RISC-Microprozessoren. *Informatik-Spektrum*, 20(5):279–267, 1997.

[30] Thomas Erlebach. Automatische Parallelisierung von Divide-and-Conquer-Algorithmen. Master's thesis, Technische Universität München, München, Germany, August 15, 1994.

[31] Thomas Erlebach. *APRIL 1.0 User Manual – Automatic Parallelization of Divide and Conquer Algorithms*. Technische Universität München, 1995.

[32] Thomas Fahringer. Using the P3T to guide the parallelization and optimization effort under the Vienna Fortran compilation system. Technical Report TR 94-5, University of Vienna, April 1994.

[33] Jerome A. Feldman, Chu-Cheow Lim, and Thomas Rauber. The shared-memory language pSather on a distributed-memory multiprocessor, 1991.

[34] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[35] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth ACM Symposium on the Theory of Computing*, pages 114–118, 1978.

[36] High Performance Fortran Forum. *High Performance Fortran Language Specification 2.0*, January 1997.

[37] Jerome H. Friedman. Multivariate adaptive regression splines (with discussion). *Annals of Statistics*, 19:1–141, 1991.

[38] David Gelernter. Parallel programming in linda. Technical Report 359, Yale University Department of Computer Science, New Haven, CT, USA, January 1985.

[39] Steve Gregory. *Parallel Logic Programming in PARLOG, The Language and Its Implementation*. Addison-Wesley, Reading, MA, USA, 1987.

[40] Steve Gregory. Experiments with speculative parallelism in PARLOG. In D. Miller, editor, *Proceedings of the 10th International Symposium on Logic Programming*. MIT press, 1993.

[41] Martin Haas and Werner Zorn. *Methodische Leistungsanalyse von Rechnersystemen*, volume 2.6 of *Handbuch der Informatik*. R.Oldenbourg Verlag, München, Germany, 1995.

[42] Per Brinch Hansen. Efficient parallel recursion. *ACM SIGPLAN Notices*, 30(12):9–16, December 1995.

[43] Stefan U. Hänßgen. REAPAR User Manual and Reference: Automatic Parallelization of Irregular Recursive Programs. Technical Report 8/98, Universität Karlsruhe, Fakultät für Informatik, Karlsruhe, Germany, March 1998.

[44] Jonathan C. Hardwick. An efficient implementation of nested data parallelism for irregular divide-and-conquer algorithms. In *First International Workshop on High-Level Programming Models and Supportive Environments*, pages 105–114, April 1996.

[45] Justiani Hendren and Laurie J. Hendren. Supporting array dependence testing for an optimizing/parallelizing C compiler. Technical report, School of Computer Science, McGill University, Montreal, Quebec, Canada, 1992.

[46] Laurie J. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell University, Ithaca, NY, USA, April 1990.

[47] James E. Hicks, Mark H. Nodine, Michael J. Beckerle, and Cotton Seed. IPC: Implicitly parallel C. Technical Report MCRC-TR-44, Motorola Cambridge Research Center, Cambridge, MA, USA, January 30, 1995.

[48] W. Daniel Hillis. *The Connection Machine*. PhD thesis, MIT, ACM distinguished dissertations, 1985.

[49] W. Daniel Hillis and Guy L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[50] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, USA, June 1994. ACM.

[51] Holger Hopp, Daniela Genius, and Michael Philippsen (Hrsg.). Seminar-Beiträge Cache-Optimierungen. Technical Report 5/98, Universität Karlsruhe, Fakultät für Informatik, Karlsruhe, Germany, January 1998.

[52] Holger Hopp and Lutz Prechelt. CuPit-2: A portable parallel programming language for artificial neural networks. In A. Sydow, editor, *Proc. 15th IMACS World Congress on Scientific Computation, Modelling, and Applied Mathematics*, pages 493–498, Berlin, Germany, August 1997. Wissenschaft & Technik Verlag.

[53] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, 1978.

[54] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 218–229, Orlando, FL, USA, June 1994. ACM.

[55] Y.-S. Hwang, B. Moon, Shamik D. Sharma, Raja Das, and Joel Saltz. Runtime support to parallelize adaptive irregular programs. In *Proceeding of the Workshop on Environments and Tools for Parallel Scientific Computing*, May 1994.

[56] Jordi Petit i Silvestre and Thomas Römke. Programming Frames for the efficient use of parallel systems. Technical Report PC^2 Technical Report TR-001-97, Paderborn Center for Parallel Computing, Paderborn, Germany, January 1997. Submitted to EUROPAR'97.

[57] Christopher F. Joerg. *The Cilk System for Parallel Multithreaded Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1996.

[58] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook.* MIT Press, 1994.

[59] Rita Loogen. *Parallele Implementierung funktionaler Programmiersprachen.* Informatik-Fachberichte 232. Springer-Verlag, Heidelberg, Germany, 1990.

[60] Welf Löwe. Optimizatiom of PRAM-programs with input-dependent memory access. In *EUROPAR 95*, Lecture Notes in Computer Science, 1995.

[61] Welf Löwe. *Optimierung paralleler Programme.* PhD thesis, Fakultät für Informatik, Universität Karlsruhe, May 29 1996.

[62] Welf Löwe and Martin Trapp. parSather: A parallel Sather. Report 95, Universität Karlsruhe, Fakultät für Informatik, 1995.

[63] Welf Löwe and Wolfgang Zimmermann. Upper time bounds for executing PRAM-programs on the LogP-machine. In *Proceedings of the 9th ACM International Conference on Supercomputing*, 1995.

[64] Steven S. Lumetta, Arvind Krishnamurthy, and David E. Culler. Towards modeling the performance of a fast connected components algorithm on parallel machines. In *Supercomputing 1995 Proceedings*, San Diego, CA, USA, December 3–8, 1995. ACM.

[65] Steven S. Lumetta, L. Murphy, X. Li, D. Culler, and I. Khalil. Decentralized optimal power pricing: The development of a parallel program. Technical report, Department of Computer Science, UCSB, Berkeley, 1995.

[66] Benoit B. Mandelbrot. *The Fractal Geometry of Nature.* W. H. Freeman and Company, New York, USA, 1983.

[67] Wellington Santos Martins. Parallel implementation of functional languages. In Herbert Kuchen and Rita Loogen, editors, *4th International Workshop on the Parallel Implementation of Functional Languages*, Aachen, Germany, September 1992. University of East Anglia, Norwich, UK.

[68] D. Michie, D.J. Spiegelhalter, and C.C. Taylor, editors. *Machine Learning, Neural and Statistical Classification.* Ellis Horwood, 1994.

[69] Jayadev Misra. Powerlist: A structure for parallel recursion. Technical report, Department of Computer Science, University of Texas at Austin, May 11, 1993.

[70] Peter Møller-Nielsen and Jørgen Staunstrup. Problem-heap. a paradigm for multiprocessor algorithms. *Parallel Computing*, 4:63–74, 1987.

[71] Zhijing George Mou. Divacon: A parallel language for scientific computing based on divide-and-conquer. In *Frontiers'90*, pages 451–461. IEEE, October 1990.

[72] Zhijing George Mou. *A Formal Model for Divide-and-Conquer and Its Parallel Realization*. PhD thesis, Yale University, May 1990.

[73] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers, and Joel Saltz. Efficient support for irregular applications on distributed-memory machines. In *PPoPP'95*, July 1995.

[74] Susumu Nishimura and Atsushi Ohori. A calculus for exploiting data parallelism on recursively defined data. In *Proceedings of the International Workshop on Theory and Practice of Parallel Programming*. (LNCS 907), 1994.

[75] Mark H. Nodine, James E. Hicks, Cotton Seed, and Michael J. Beckerle. Generating parallelism profiles from C programs. Technical Report MCRC-TR-43, Motorola Cambridge Research Center, Cambridge, MA, USA, September 16, 1004.

[76] Daniel W. Palmer, Jan F. Prins, and Stephen Westfold. Work-efficient nested data-parallelism. In *Frontiers'95*, McLean, Virginia, USA, February 6–9, 1995. IEEE.

[77] Michael Philippsen. *Optimierungstechniken zur Übersetzung paralleler Programmiersprachen*. Fortschrittberichte 10/292. VDI Verlag, Düsseldorf, 1994.

[78] Andrew James Piper. *Object-Oriented Divide-and-Conquer for Parallel Processing*. PhD thesis, Emmanuel College, Cambridge, England, July 1994.

[79] Andrew James Piper and R.W. Prager. Generalized parallel programming with divide-and-conquer: The beeblebrox system. Technical Report CUED/F-INFENG/TR132, Cambridge University Engineering Department, Cambridge, England, July 1993.

[80] Carl Edward Rasmussen, Radford M. Neal, Geoffrey Hinton, Drew van Camp, Michael Revow, Zoubin Ghahramani, Rafal Kustra, and Rob Tibshirani. Delve - data for evaluating learning in valid experiments. `http://www.cs.utoronto.ca/~delve/`, 1998.

[81] Niels Reimer, Stefan U. Hänßgen, and Walter F. Tichy. Dynamically adapting the degree of parallelism with reflexive programs. In *Proceedings of the Third International Workshop on Parallel Algorithms for Irregularly Structured Problems (IRREGULAR), Springer LNCS 1117*, pages 313–318, Santa Barbara, CA, USA, August 19–21, 1996.

[82] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM TOPLAS*, 19(6):942–991, nov 1997.

[83] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. Supporting dynamic data structures on distributed memory machines. In *TOPLAS Proceedings*. ACM, March 1995.

[84] Robert Sedgewick. *Algorithms*. Computer Science. Addison-Wesley, 2nd edition edition, 1988.

[85] Jaswinder Pal Singh. *Parallel Hierarchical N-Body methods and their implications for parallel processors*. PhD thesis, Stanford University, February 1993.

[86] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-92-526, CSL, Stanford University, Stanford, CA, USA, June 1992.

[87] Thomas Stirner. Laufzeitoptimierung durch Tuning anhand geeigneter Beispielprogramme. Master's thesis, Institut Rechnersysteme, TU Dresden, Dresden, Germany, March 8, 1995.

[88] David Stoutamire. The pSather 1.0 manual. Technical Report TR-95-058, International Computer Science Institute, Berkeley, CA, USA, October 1995.

[89] SunSoft, Mountain View, CA, USA. *pthreads and Solaris threads: A comparison of two user level thread APIs*, early access edition, May 1994.

[90] SunSoft, Mountain View, CA, USA. *Solaris 2.4 Multithreaded Programming Guide*, August 1994.

[91] Supercomputing Technologies Group, MIT Laboratory for Computer Science, Cambridge, MS, USA. *Cilk 5.0 (Beta 1) Reference Manual*, March 1997.

[92] Boreslaw K. Szymanski, editor. *Parallel Functional Languages and Compilers*. Frontier Series. ACM Press, New York, NY, USA, 1991.

[93] Thomas M. Warschko. *Effiziente Kommunikation in Parallelrechnerarchitekturen*. PhD thesis, Institut für Programmstrukturen und Datenorganisation, Fakultät für Informatik, Universität Karlsruhe, Karlsruhe, Germany, 1997. To appear.

[94] Andreas Wehrenpfennig. *Programmiereinführung in OPAL*. Fakultät für Informatik, Technische Universität Dresden, Dresden, Germany, March 20, 1997.

[95] Michael J. Wolfe. *High Performance Compilers for Parallel Computers*. Addison-Wesley, Redwood City, CA, USA, 1996.