View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by KITopen

Automatische Einstellung des Parallelitätsgrades von Programmen

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

der Fakultät für Informatik

der Universität Karlsruhe (Technische Hochschule)

genehmigte

Dissertation

von

Otilia Werner-Kytölä

aus São Sebastião do Caí, Brasilien

Tag der mündlichen Prüfung: 19. Oktober 1999

Erster Gutachter: Prof. Dr. Walter F. Tichy
Zweiter Gutachter: Prof. Dr. Theo Ungerer

Danksagung

Herr Prof. Tichy, vielen Dank für die Übernahme des Hauptreferats und für die Gelegenheit, an Ihrem Lehrstuhl meine Doktorarbeit durchführen zu dürfen. Ferner möchte ich mich bei Ihnen für die zahlreichen fruchtbaren Diskussionen in allen Phasen meiner Arbeit und für die kritische Durchsicht meiner Dissertation bedanken.

Herr Prof. Ungerer, vielen Dank für die Übernahme des Korreferats, für das Interesse an meiner Arbeit sowie für die wertvollen Verbesserungsvorschläge.

Michael, vielen Dank für das kritische Durchlesen in der früheren Phase des Zusammenschreibens.

Matthias, vielen Dank für die vielen fachlichen Diskussionen und für die gemeinsame Zeit auf der „Welt der Cray T3E“.

Timo, vielen, vielen Dank für alles.

Barbara, vielen Dank für das Zusammenhalten in allen diesen Jahren und auch für alle die „Wie-sagt-man-das-auf-Deutsch“-Stunden.

Pai, muito obrigada pelo incentivo em todos estes anos e pela regra do " não te mixa".

Olga, muito obrigada pelo apoio desde o início da minha chegada à Alemanha e pela eterna torcida para que tudo desse certo!

Diana, Soraia, Sonc, Schultz, Val, Lu, Rafa, valeu pela forca!

CNPq, muito obrigada pelo auxílio financeiro durante quatro anos do meu doutorado no exterior.

Otilia Werner-Kytölä　　　　Karlsruhe, im Oktober 1999.

Inhaltsverzeichnis

Danksagung I

Zusammenfassung VI

1 Einführung 1
1.1 Motivation . . . 1
1.2 Ziel der Dissertation . . . 3
1.3 Beiträge dieser Arbeit . . . 3
1.4 Aufbau der Arbeit . . . 5

2 Grundlagen 7
2.1 Parallelrechner . . . 7
2.2 Architekturmodelle . . . 7
2.3 Parallele Programmiermodelle . . . 9
2.4 Charakterisierung paralleler Programme . . . 11
2.5 Modellierung der Leistung paralleler Programme . . . 15
2.5.1 Das Amdahl'sche Gesetz . . . 15
2.5.2 Asymptotische Analyse . . . 16
2.5.3 Planbare Algorithmen . . . 17
2.5.4 Methode nach Foster . . . 17
2.6 Entwurf paralleler Software . . . 18

3 Stand der Forschung 21
3.1 Verbesserung der Auslastung von Parallelrechnern . . . 21
3.2 Verbesserung der Leistung paralleler Programme . . . 25
3.3 Kombinierte Verbesserung der Auslastung und Leistung . . . 26
3.4 Einordnung und Vergleich der Arbeiten . . . 27

4 Theoretische Modellbildung 29
4.1 Methode . . . 29
4.2 Klassifizierung von Programmen . . . 32
4.3 Analyse von Programmklassen . . . 32
4.3.1 Analyse in Hinblick auf die Ausführungszeit . . . 33
4.3.2 Analyse in Hinblick auf die Effizienz . . . 35
4.3.3 Analyse in Hinblick auf das Nutzen-Kosten-Verhältnis . . . 37
4.4 Das Amdahl'sche Gesetz im Nutzen-Kosten-Verhältnis . . . 41
4.5 Diskussion der theoretischen Modellbildung . . . 42

5 Automatische Bestimmung des Parallelitätsgrades 45
5.1 Randbedingungen und Aufgaben 45
5.2 Szenario der Einstellung des Parallelitätsgrades 48
5.3 Festlegung der Stellen für die Einstellung 49
5.4 Suchverfahren zur Parallelitätsgradbestimmung 50
5.5 Strategien für die Einstellung des Parallelitätsgrades 50
5.5.1 Lokale Strategien 51
5.5.2 Globale Strategien 54
5.5.3 Übersicht über die Strategien 55
5.6 Laufzeitinformationen und Messungen 55
5.7 Datenumverteilung 56
5.7.1 Grundidee 56
5.7.2 Arten der Datenverteilung 57
5.7.3 Unterstützte Datenstrukturen 58
5.8 Arten der Einstellung 59
5.9 Parallelitätsgrad-Datenbank 61

6 Realisierung 63
6.1 Szenario der Einstellung des Parallelitätsgrades 63
6.2 Einschränkungen 63
6.3 Verwendete Werkzeuge 64
6.4 Prozeßarchitektur 65
6.5 Klassendiagramm 66
6.6 Schnittstelle zwischen Benutzer und System 68
6.7 Instrumentierung 68
6.7.1 Spezifizierung der Stellen für die Einstellung 69
6.7.2 Spezifizierung der Parameter 70
6.7.3 Änderungen im Programm 72
6.8 Verwaltung der Einstellung des Parallelitätsgrades 73
6.8.1 Grundidee 73
6.8.2 Intra-Lauf-Einstellung 74
6.8.3 Inter-Lauf-Einstellung 75
6.9 Einstellung des Parallelitätsgrades 77
6.9.1 Grundidee 77
6.9.2 Algorithmen 77
6.10 Datenumverteilung 78
6.10.1 Grundidee 78
6.10.2 Datenaustausch zwischen *Worker* 79
6.10.2.1 Export-Operation 80
6.10.2.2 Import-Operation 81
6.11 Ausgabe der Einstellung 82

7 Ergebnisse 87
7.1 Benchmark-Sammlung 87
7.1.1 Radix-Sortieralgorithmus 87
7.1.2 *Livermore Loop 1* (LL1) 89
7.1.3 *Livermore Loop 3* (LL3) 90
7.1.4 *Livermore Loop 13* (LL13) 90

7.1.5 Veltran-Operator . . . 91
7.1.6 TLM-Verfahren . . . 92
7.1.7 PDE-Löser . . . 93
7.2 Einstellung über das mathematische Modell . . . 94
7.2.1 Validierungstechnik . . . 95
7.2.2 Programmklassen der Benchmark-Sammlung . . . 95
7.2.3 Diskussion der Ergebnisse . . . 96
7.2.3.1 LL1 . . . 97
7.2.3.2 Radix-Sortieralgorithmus . . . 98
7.2.3.3 LL13 . . . 99
7.2.3.4 TLM-Verfahren . . . 100
7.2.3.5 PDE-Löser . . . 101
7.2.3.6 LL3 . . . 102
7.2.3.7 Veltran-Operator . . . 103
7.2.4 Genauigkeit des mathematischen Modells . . . 104
7.2.5 Extrapolation aus den erzielten Ergebnissen . . . 105
7.3 Einstellung über Suchverfahren . . . 106
7.3.1 Validierungstechnik . . . 106
7.3.2 Mehraufwand der Instrumentierung . . . 106
7.3.3 Zur Präsentation der Ergebnisse . . . 107
7.3.4 Diskussion der Ergebnisse . . . 108
7.3.4.1 Radix-Sortieralgorithmus . . . 110
7.3.4.2 *Livermore Loop 1* (LL1) . . . 114
7.3.4.3 *Livermore Loop 3* (LL3) . . . 115
7.3.4.4 *Livermore Loop 13* (LL13) . . . 118
7.3.4.5 Veltran-Operator . . . 120
7.3.4.6 TLM-Verfahren . . . 124
7.3.4.7 PDE-Löser . . . 127
7.3.5 Genauigkeit des Suchverfahrens . . . 132
7.3.6 Extrapolation aus den erzielten Ergebnissen . . . 133
7.4 Vergleich der zwei Methoden . . . 134

8 Zusammenfassung und Ausblick **139**
8.1 Beiträge dieser Arbeit . . . 139
8.2 Ausblick auf zukünftige Forschung . . . 140

A **143**

Anhang **143**

Literaturverzeichnis **145**

Zusammenfassung

Einem parallelen Programm werden oft alle zur Verfügung stehenden Prozessoren eines Multiprozessorrechners zugeteilt. Diese Technik kann aber zu schlechten Ergebnissen führen, weil sie davon ausgeht, daß alle Prozessoren ausgenutzt werden können, was der Realität nicht unbedingt entspricht. Der Benutzer läßt auf dieser Weise seine Programme u.U. auf einer zu hohen Anzahl von Prozessoren laufen, wodurch sich die Ausführungszeit des Programmes durch die zusätzlichen Kommunikationskosten verlängert oder eine schlechte Auslastung des Rechners resultiert.

Diese Arbeit beschäftigt sich mit der automatischen Einstellung des Parallelitätsgrades von Programmen, wobei die optimale Anzahl von Prozessoren für ein paralleles Programm gesucht wird. Die Einstellung des Parallelitätsgrades verfolgt zwei Ziele:

1. Das Laufzeitverhalten, nämlich die Ausführungszeit, die Effizienz oder die Kosten-Nutzen-Relation von parallelen Programmen, die für ein mit verteiltem Speicher realisiertes Mehrprozessorsystem entwickelt wurden, soll verbessert werden. Die Verbesserung des Laufzeitverhaltens wird gegenüber der Ausführung des Programmes ohne Einstellung des Parallelitätsgrades gemessen, d.h. der Ausgangspunkt ist die Ausführung mit der vom Benutzer oder vom Betriebssystem ausgewählten Anzahl von Prozessoren.

2. Die Auslastung des Gesamtrechners soll verbessert werden. Einem parallelen Programm wird durch die Einstellung des Parallelitätsgrades eine maximale Prozessoranzahl zugeteilt, bei der die Effizienz nicht unterhalb einer vorgegebenen Schranke abfällt, oder so viele Prozessoren, daß die Kosten-Nutzen-Relation ihren minimalen Wert erreicht. Folge daraus ist, daß so viele Prozessoren wie möglich für die Ausführung weiterer, in das Mehrprozessorsystem kommender Programme freibleiben, was dem Benutzer letztendlich zugute kommt, da er dadurch z.B. mehrere Programme gleichzeitig laufen lassen kann.

In dieser Arbeit wurden zwei Methoden zur Bestimmung des optimalen Parallelitätsgrades entwickelt. Bei der ersten wird der Parallelitätsgrad durch ein mathematisches Modell ermittelt, während er bei der zweiten automatisch anhand eines Optimierungsverfahrens bestimmt wird.

Über das mathematische Modell wird der optimale Parallelitätsgrad anhand der Ausführungszeit, Effizienzschranke sowie Nutzen-Kosten-Relation approximiert, vorausgesetzt, der Kommunikations- und der Berechnungsaufwand der Programme sind bekannt. Diese Methode wurde auf sieben parallele Programme angewendet, wobei der Fehler im Durchschnitt ca. 13% betrug.

Bei der automatischen Einstellung des Parallelitätsgrades wird die Anzahl der in der Ausführung eines parallelen Programmes beteiligten Prozessoren solange nach einer gezielten Strategie verändert, bis das Programm nach angegebenen Kriterien optimal läuft. Die Einstellung wird entweder *intra-Lauf* oder *inter-Lauf* durchgeführt. Die erste findet innerhalb einer Ausführung des Programmes statt, während die zweite zwischen mehreren Ausführungen erfolgt. Bei den beiden Arten der Einstellung wird der Parallelitätsgrad anhand von Informationen eingestellt, die während der Laufzeit des Programmes aufgenommen werden.

Die Methode zur automatischen Einstellung des Parallelitätsgrades wurde für die Cray T3E implementiert und an drei realen Anwendungen, drei der *Livermore Loops* sowie an einem Sortieralgorithmus bewertet. Die drei realen Anwendungen sind ein Operator aus der Geophysik, das TLM(*Transmission Line Matrix*)-Verfahren aus der Elektrotechnik und ein Löser partieller Differentialgleichungen (PDE). Als Ergebnis stieg die Effizienz bei dem geophysikalischen Operator um bis zu 24% bei einer vorgegebenen Effizienzschranke von 80%. Die Kosten-Nutzen-Relation hat ihr Minimum für das TLM-Verfahren erreicht, wobei sie um Faktor 2 bei der Einstellung des Parallelitätsgrades erniedrigt wurde. Bei der PDE-Anwendung wurde der Parallelitätsgrad, bei dem das Kosten-Nutzen-Verhältnis aufs Minimum gesetzt wurde, auf 16 Prozessoren eingestellt, wobei eine Inter-Lauf-Einstellung benutzt wurde. Bei dem Sortieralgorithmus wurde eine Reduktion der Ausführungszeit um bis zum Faktor 3,5 erzielt.

Kapitel 1

Einführung

Die Parallelrechnerei etabliert sich mehr und mehr als die effektive Lösung, wenn nach Rechenleistung gefragt wird. Seitens der Hardware bedeutet es, über eine Architektur zu verfügen, die aus vielen, gleichzeitig zu benutzenden Prozessoren besteht. Seitens der Software bedeutet es, ein Programm so zu strukturieren, daß es aus einer Menge unabhängiger Aktivitäten besteht.

Wenn man sich von der sequentiellen zu der parallelen „Welt“ bewegt, ist eine Verbesserung der Leistung von Programmen bzw. die nicht-Verschlechterung der Auslastung des Gesamtsystems sowohl wegen der Hardware- als auch wegen der Software-Domäne nicht selbstverständlich. Beeinträchtigende Faktoren sind die Konkurrenz um gemeinsame Betriebsmittel, die zusätzlichen Kosten, die durch die Kommunikation zwischen Prozessoren entstehen, und die begrenzte Anzahl paralleler Aktivitäten, die das Programm anzubieten hat. Das Gewicht aller dieser Faktoren kann aber drastisch reduziert werden, wenn einem Programm nur so viele Prozessoren zugeordnet werden, wie es sinnvollerweise beschäftigen kann. Damit befaßt sich die vorliegende Arbeit.

1.1 Motivation

Sequentielle Programme werden parallelisiert, um ihre Ausführungszeiten zu verkürzen. Dieses Ziel wird jedoch oft nicht erreicht, weil der Parallelitätsgrad überdimensioniert wird. Es gibt immer einen Punkt, ab dem jeder zusätzliche, einem Programm zugeordnete Prozessor nicht mehr zu einer Verbesserung der Leistung beiträgt bzw. sogar zu einer Verschlechterung derselben führen kann. Gründe dafür sind einerseits die steigenden Kommunikationskosten, andererseits der Mangel an parallelen Aktivitäten seitens des Programmes. Daher ist die Bestimmung des Sättigungspunktes bzw. des optimalen Parallelitätsgrades des Programmes ein erstrebenswertes Ziel.

Ohne Unterstützung seitens des Systems nimmt der Benutzer normalerweise an, daß sein Programm besser parallelisierbar sei als es tatsächlich ist, und belegt dafür zu viele Prozessoren. Genauso belegt das Betriebssystem eines Mehrprozessorsystems häufig alle noch zur Verfügung stehenden Prozessoren für die Ausführung eines Programmes, ohne zu berücksichtigen, ob sich das überhaupt lohnt. Aus diesen Gründen ist eine automatische Bestimmung des optimalen Parallelitätsgrades eines Programmes sinnvoll.

Die Parallelisierung von Programmen geschieht immer häufiger automatisch. Systeme, die solche Programme instrumentieren bzw. umschreiben, liefern aber keinerlei Information über die Anzahl der zu belegenden Prozessoren, die z.B. zu einer erwünschten Beschleunigung führen würde. Diese Systeme können erfolgreich die parallelisierbaren Aktivitäten

in einem Programm identifizieren, sind aber mangelhaft in der Bestimmung des optimalen Parallelitätsgrades für ein Programm.

Ein weiteres zu berücksichtigendes Phänomen ist, daß der Parallelitätsgrad eines Programmes normalerweise in der Zeit variiert [28]. Die Folge daraus ist, daß auch, wenn z.B. das Programm mit einer geeigneten Anzahl von Prozessoren gestartet wurde, diese während der Ausführung des Programmes suboptimal werden kann. Dies geschieht dann, wenn das Programm in einer bestimmten Ausführungsphase weniger Parallelismus anzubieten hat oder wenn das Verhältnis zwischen Kommunikations- und Berechnungsaufwand zu hoch wird. Die Einstellung des Parallelitätsgrades ist auch daher während der Ausführung des Programmes anhand von Laufzeitinformationen notwendig.

Die Anpassung des Parallelitätsgrades ist nicht nur für das jeweils betrachtete Programm sehr wichtig, sondern auch für den Gesamtrechner. Es ist immer wünschenswert, daß das System gut ausgelastet bleibt. Wenn einige Prozessoren, die der Ausführung eines parallelen Programmes zugeordnet wurden, nicht mehr gut ausgelastet sind, sollten sie für die Ausführung anderer, in das Mehrprozessorsystem kommender Programme freigestellt werden. Das Erreichen einer guten Auslastung des Mehrprozessorsystems kommt auch dem Benutzer zugute. Im besten Fall wird die Ausführungszeit seines Programmes durch die Reduzierung der Anzahl von Prozessoren wegen der daraus entstandenen Erniedrigung der Kommunikationskosten verkürzt. Wenn das aber nicht der Fall ist, kann die Verbesserung der Auslastung dem Benutzer immer noch von Vorteil sein, indem die Gesamtausführungszeit mehrerer von ihm gleichzeitig gestarteter Programme reduziert wird.

Als Beispiel betrachten wir ein fiktives Programm A, das auf einem Parallelrechner ausgeführt wird, der insgesamt über 8 Prozessoren verfügt. Die Ausführungszeit des Programmes auf 4 Prozessoren (33s) ist 32% höher als auf 8 Prozessoren (25s), seine Effizienz steigt aber um 50%, wenn der Parallelitätsgrad von 8 auf 4 reduziert wird. Jedoch ist die benötigte Zeit für zwei aufeinanderfolgende Ausführungen dieses Programmes auf 4 Prozessoren 51% kleiner als wenn 8 Prozessoren eingesetzt werden, wie aus der Tabelle 1.1 unter *Gesamtzeit* herauszulesen ist. Dies liegt daran, daß beim Parallelitätsgrad 4 die zwei Ausführungen gleichzeitig erfolgen können, und beim Parallelitätsgrad 8 zwei aufeinanderfolgende Ausführungen benötigt werden. Bei 8 Läufen des Programmes liefert sogar ein Parallelitätsgrad von 1 die minimale Ausführungszeit, wobei sie 32% bzw. um Faktor 4 niedriger ist als beim Parallelitätsgrad 4 bzw. 8.

Tabelle 1.1: Kennzahlen des fiktiven Programmes A

Parallelitätsgrad	Ausführungszeit (s) (1 Programmlauf)	Beschleunigung	Effizienz	Gesamtzeit (s) (2 Programmläufe)
1	100,0	1,00	1,00	100,0
4	33,0	3,03	0,75	33,0
8	25,0	4,00	0,50	50,0

Ein weiteres Argument, warum eine gute Auslastung des Gesamtsystems auch dem Benutzer von Interesse ist, sind die bei der Benutzung mehrerer Prozessoren entstehenden Kosten. Der Benutzer eines Mehrprozessorsystems muß dem Betreiber für den Zugriff auf das System bezahlen, wobei sich die Kosten üblicherweise auf die benutzten Zeiteinheiten pro Prozessor beziehen. Wenn nur so viele Prozessoren alloziert werden wie die Program-

me des Benutzers gut auslasten können, werden den Programmen evtl. weniger Prozessoren zugeordnet. Daraus folgt, daß die Kosten für die Benutzung des Mehprozessorsystems ggf. verringert werden können. Auf der anderen Seite gibt es Rechenzentren, die eine ineffiziente Benutzung der zur Verfügung gestellten Resourcen bestrafen. Beispielsweise belastet ein Rechenzentrum Benutzer, die die Vektoreinheiten eines Vektorrechners kaum benutzen, viermal soviel wie Benutzer, die von den Vektoreinheiten intensiv Gebrauch machen. Auch in dieser Situation ist es wichtig, daß der Benutzer auf die Effizienz achtet.

Zusammenfassend läßt sich sagen, daß die automatische Einstellung des Parallelitätsgrades sinnvoll erscheint. Neben der Verbesserung des Laufzeitverhalten eines Programmes ist bei der Anpassung des Parallelitätsgrades ein weiteres Ziel von Interesse, nämlich das Erreichen einer guten Auslastung des Mehrprozessorsystems.

1.2 Ziel der Dissertation

Diese Dissertation verfolgt zwei Thesen:

These 1: *Es ist möglich, den Parallelitätsgrad eines Programmes dynamisch einzustellen und eine Verbesserung des Laufzeitverhaltens von parallelen Programmen zu erreichen, die für ein mit verteiltem Speicher realisiertes Mehrprozessorsystem entwickelt wurden. Je nach Ziel der Einstellung entspricht das Laufzeitverhalten der Ausführungszeit, der Effizienz oder dem Kosten-Nutzen-Verhältnis. Die Verbesserung des Laufzeitverhaltens wird gegenüber der Ausführung des Programmes ohne Einstellung des Parallelitätsgrades gemessen, d.h. der Ausgangspunkt ist die Ausführung mit der vom Benutzer oder Betriebssystem ausgewählten Anzahl von Prozessoren.*

These 2: *Die Auslastung des Gesamtrechners kann durch die Einstellung des Parallelitätsgrades von Programmen verbessert werden. Einem parallelen Programm wird durch die Einstellung des Parallelitätsgrades eine maximale Prozessoranzahl zugeteilt, bei der die Effizienz eine vorgegebene Schranke nicht unterschreitet, oder so daß die Kosten-Nutzen-Relation ihren minimalen Wert erreicht. Folge daraus ist, daß so viele Prozessoren wie möglich für die Ausführung weiterer, in das Mehrprozessorsystem kommender Programme freibleiben, was dem Benutzer letztendlich zugute kommt.*

Unter Kosten-Nutzen-Relation versteht man das Verhältnis zwischen den Kosten, die bei der Benutzung von mehr als einem Prozessor für die Ausführung eines Programmes entstehen, und dem Nutzen, der wiederum dem Verhältnis zwischen der Ausführungszeit des Programmes mit einem Prozessor und mit p Prozessoren entspricht, wobei $p > 1$ gilt.

Das Ziel der vorliegenden Dissertation ist es, die o.g. Thesen zu belegen. Dafür wird ein Modell zur Bestimmung des optimalen Parallelitätsgrades entwickelt und anhand eines Experimentiersystems sowie realer Anwendungen validiert.

1.3 Beiträge dieser Arbeit

Die vorliegende Arbeit ist die erste, die sich systematisch mit der automatischen Einstellung des Parallelitätsgrades auf Rechnern mit verteiltem Speicher beschäftigt hat. Die zwei präsentierten Thesen werden belegt, d.h. das Laufzeitverhalten paralleler Programme bzw.

die Auslastung des Gesamtrechners kann verbessert werden. Diese Aussage basiert auf der Untersuchung von drei realen Anwendungen, einem Sortieralgorithmus und drei der Livermore Loops.

Ein mathematisches Modell für die Berechnung des optimalen Parallelitätsgrades wurde entwickelt und anhand einer Benchmark-Sammlung mit sieben Programmen validiert. Das mathematische Modell ist fähig, den nach der Ausführungszeit, Effizienz sowie der Kosten-Nutzen-Relation optimalen Parallelitätsgrad zu approximieren. Bis auf einige Situationen, die in Abschnitt 7.2.3 beschrieben werden, bleibt die Auswirkung der Abweichung zwischen dem gemessenen und dem berechneten optimalen Parallelitätsgrad gering, nämlich im Durchschnitt ca. 13%.

Effiziente Strategien für die Optimierung unterschiedlicher Kennzahlen, wie die Ausführungszeit bzw. die Beschleunigung paralleler Programmen, die Effizienz oder die Kosten-Nutzen-Relation, werden entwickelt, implementiert und getestet. Insgesamt handelt es sich um acht Strategien, wobei drei komplett neu entwickelt wurden und fünf auf [45] basierend angepaßt bzw. weiterentwickelt wurden.

Ein Mechanismus für die Datenumverteilung verschiedener Datenstrukturen wurde entwickelt, implementiert und getestet. Die Datenumverteilung wurde so konzipiert, daß sie die Umverteilung unterschiedlicher Strukturen unterstüzt und auch einen Mechanismus vorsieht, wodurch der Benutzer leicht die Umverteilung neuer Datenstrukturen implementieren kann.

Es wurden zwei Arten der Einstellung des Parallelitätsgrade entwickelt, nämlich die *Intra-Lauf-* und die *Inter-Lauf*-Einstellung. Die beiden Möglichkeiten stehen dem Benutzer mittels der Bibliotheksroutinen zur Verfügung. Die erste Art der Einstellung findet innerhalb einer Ausführung des Programmes statt, während die zweite zwischen mehreren Läufen des Programmes geschieht. Vorteil der Inter-Lauf-Einstellung ist, daß dabei keine Datenumverteilung vorgenommen werden muß. So kann der Parallelitätsgrad auch von solchen Programmen eingestellt werden, deren Intra-Lauf-Einstellung wegen der Datenumverteilung mit sehr hohen Kosten verbunden wäre.

Im Rahmen dieser Arbeit wurde auch das System *ParGrad* entwickelt, das die automatische Einstellung des Parallelitätsgrades von in C++ geschriebenen Programmen auf einer Cray T3E, einem Parallelrechner mit verteiltem Speicher, realisiert. Das System steht dem Benutzer als eine Bibliothek zur Verfügung. Der Benutzer instrumentiert sein paralleles Programm durch das Einfügen von Aufrufen von Bibliotheksroutinen und die Spezifizierung einiger Parameter, wie die Einstellungsstrategie oder die umzuverteilenden Datenstrukturen. Nach der Instrumentierung läuft das Programm mit der Einstellung des Parallelitätsgrades. Drei Anwendungen, ein Sortieralgorithmus und drei Kerne wurden mit dem System *ParGrad* auf die Einstellung des Parallelitätsgrades hin untersucht. Die Anwendungen sind ein Operator aus der Geophysik, das TLM(*Transmission Line Matrix*)-Verfahren, ein Löser partieller Differentialgleichungen (PDE, *Partial Differential Equation*) und ein Sortieralgorithmus. Bei dem Sortieralgorithmus wurde eine Reduzierung der Ausführungszeit bis zum Faktor 3,5 erzielt. Die Effizienz wurde bei dem geophysikalischen Operator um bis zu 24% höher. Das Kosten-Nutzen-Verhältnis ist um Faktor 2 bei dem TLM-Verfahren gesunken. Bei der PDE-Anwendung konnte das Kosten-Nutzen-Verhältnis minimiert werden, wobei unter der Benutzung der Inter-Lauf-Einstellung der Parallelitätsgrad auf 16 Prozessoren eingestellt wurde.

1.4 Aufbau der Arbeit

Diese Arbeit besteht aus sieben weiteren Kapiteln. Kapitel 2 beschreibt die für die Arbeit relevanten Grundlagen von Parallelrechnern, Architekturmodellen, Programmier- und Leistungsmodellen für Parallelrechner und gibt eine Charakterisierung paralleler Programme. Im Kapitel 3 wird der Stand der Forschung vorgestellt, wobei Arbeiten aus drei Gebieten präsentiert und mit der vorliegenden Arbeit verglichen werden. Danach wird im Kapitel 4 über die theoretische Modellbildung zur Ermittlung des Parallelitätsgrades diskutiert. Im Kapitel 5 findet die Beschreibung des im Rahmen dieser Arbeit entwickelten Modells zur automatischen Bestimmung des optimalen Parallelitätsgrades eines Programmes statt. Basierend darauf befaßt sich Kapitel 6 mit der Realisierung der automatischen Einstellung des Parallelitätsgrades; das daraus entstandene System *ParGrad* wird vorgestellt. Die durch die Einstellung des Parallelitätsgrades von Programmen erzielten Ergebnisse und der Nachweis der von der vorliegenden Arbeit verfolgten Thesen werden im Kapitel 7 präsentiert. Kapitel 8 faßt nochmals die wichtigsten Ergebnisse zusammen und gibt einen Ausblick auf weitere Arbeiten.

Kapitel 2

Grundlagen

In diesem Kapitel werden die für die Arbeit relevanten Grundlagen beschrieben. Hier werden Parallelrechner bzw. die unterschiedlichen Architekturmodelle präsentiert, Programmier- und Leistungsmodelle für Parallelrechner diskutiert, eine Charakterisierung paralleler Programme vorgestellt und den Entwurf paralleler Software besprochen.

2.1 Parallelrechner

Die Leistung der schnellsten Rechner ist seit dem Jahr 1945 bis heutzutage exponentiell gewachsen [21], durschnittlich um Faktor 10 alle fünf Jahre. Auf der anderen Seite steigt die Taktrate, die die Ausführungszeit des primitivsten Befehls letzendlich bestimmt, derzeit langsam, wobei eine Tendenz zur Annäherung an die physikalische Grenze beobachtet werden kann. Diese physikalische Grenze wird durch die Lichtgeschwindigkeit vorgegeben. Um über die gegenwärtig verfügbare Einzelprozessorleistung hinauszugehen, wird die Parallelverarbeitung eingesetzt. Dafür werden verschiedene Techniken bei dem Rechnerentwurf eingeführt: „Fließband-Bearbeitung" von Befehlen (*pipelining*), Einsatz mehrerer Ausführungseinheiten (z.B. Gleitkommaeinheit) oder der Einsatz mehrerer Prozessoren. Rechner, die aus einer Menge von Prozessoren bestehen, und zur Lösung eines Problems kooperieren, nennen sich Parallelrechner.

Parallelrechner unterscheiden sich nach Flynn [19] in SIMD- und MIMD-Architekturen. Rechner, die der SIMD (*Single Instruction Multiple Data*)-Kategorie gehören, sind fähig, denselben Programmbefehl gleichzeitig bezogen auf mehrere Datensätze auszuführen. MIMD (*Multiple Instruction Multiple Data*)-Rechner dagegen bearbeiten verschiedene Teile eines Programmes auf unterschiedlichen, voneinander unabhängigen Prozessoren. Die Programmierung von Parallelrechnern zweiter Art ist aufwendiger, weil die einzelnen Prozessoren miteinander kommunizieren müssen, um die parallel zu bearbeitenden Teilaufgaben miteinander abzustimmen und den Arbeitsfortschritt zu kontrollieren [52].

Im folgenden werden verschiedene Architektur- und Programmiermodelle für Parallelrechner präsentiert.

2.2 Architekturmodelle

Die Klassifikation nach Warschko [54] berücksichtigt die logische Struktur des Speichersubsystem (verteilter oder gemeinsamer Speicher), die Definition des Adreßraumes (global

oder lokal) sowie die Kommunikationsstrukturen zwischen dem Prozessor und dem Speichersubsystem (implizite oder explizite Kommunikation). Nach dieser Klassifikation sind drei Architekturmodelle für Parallelrechner mit p Prozessoren zu erkennen:

- **Architekturen mit verteiltem Speicher** (Nachrichtenkopplung, *message passing systems*, *loosely coupled systems*) – Sind durch einen physikalisch verteilten Speicher, p prozessor-lokale Adreßräume sowie explizite Kommunikationsoperationen, die nicht auf den Speicher anderer Prozessoren zugreifen können, gekennzeichnet (siehe Abbildung 2.1).

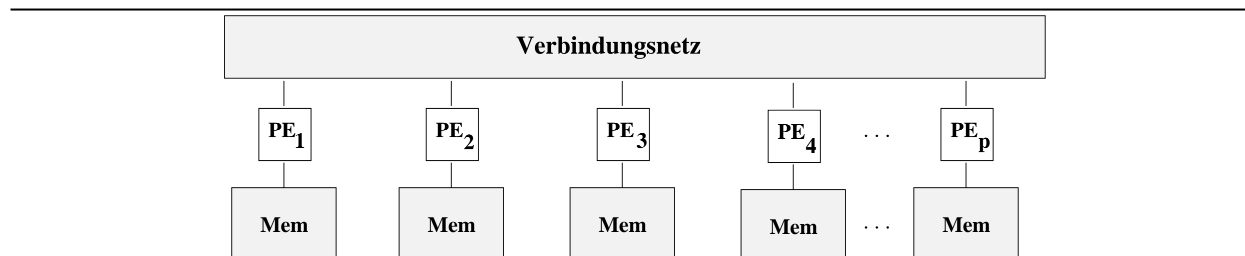


Abbildung 2.1: Architekturen mit verteiltem Speicher

- **Architekturen mit gemeinsamem Adreßraum** (*shared address space*) – Sind durch einen physikalisch verteilten Speicher, einen globalen Adreßraum sowie explizite Kommunikationsoperationen, die in Gegensatz zu den prozessoreigenen Speicherzugriffsoperationen auf den Speicher anderer Prozessoren zugreifen können, charakterisiert. Der direkte Zugriff auf den Speicher anderer Prozessoren geschieht mittels spezieller Kommunikationsoperationen, die den Schreib- bzw. Lesezugriff auf ein nicht-lokales Speicherelement realisieren, wobei das *Rendezvous*-Prinzip, das bei Systemen mit verteiltem Speicher benötigt wird, entfällt (einseitige Kommunikation). Im Gegensatz zu Systemen mit gemeinsamem Speicher ist hier der Zugriff auf nicht-lokale Datenelemente über explizite Kommunikationsoperationen unter Angabe der Prozessornummer und der Speicherstelle innerhalb des Zielprozessors realisiert.

  Für die Realisierung der vorliegenden Arbeit (siehe Kapitel 6) wurde ein Parallelrechner mit gemeinsamem Adreßraum als Zielarchitektur ausgewählt, nämlich die Cray T3E [41].

- **Architekturen mit gemeinsamem Speicher** (*shared memory systems*, *tightly coupled systems*) – Sind durch einen physikalisch gemeinsamen oder verteilten Speicher, einen globalen Adreßraum sowie implizite Kommunikationsoperationen charakterisiert, die über das Verbindungsnetzwerk (siehe Abbildung 2.2) transparent auf alle Speicherstellen zugreifen.

Das Besondere bei dieser Klassifikation im Vergleich zu der nach Hwang [29] ist die Einführung des Kriteriums Kommunikationsstrukturen, welches die eindeutige Unterscheidung zwischen Architekturen mit gemeinsamem Adreßraum und Architekturen mit gemeinsamem Speicher ermöglicht.

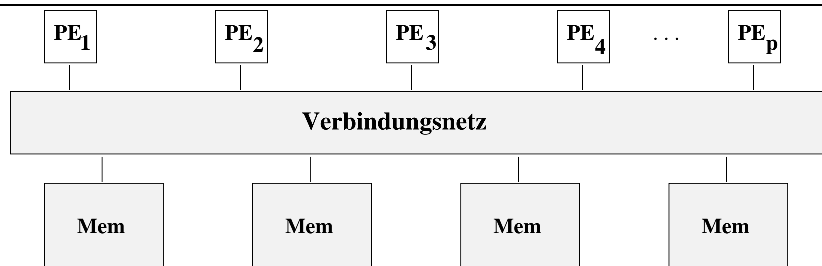


Abbildung 2.2: Architekturen mit gemeinsamem Speicher

2.3 Parallele Programmiermodelle

Die parallele Programmierung hat eine höhere Komplexität im Vergleich zur sequentiellen Programmierung: Die Zusammenarbeit einer großen Anzahl von Prozessoren muß geregelt und die Interaktion zwischen Prozessoren muß koordiniert werden. Verschiedene Modelle unterstützen die parallele Programmierung. Diese Modelle bzw. Abstraktionen unterscheiden sich in ihrer Flexibilität, dem Mechanismus zur Kommunikation zwischen Prozessen, der Unterstützung der Datenlokalität, Skalierbarkeit und Modularität [21]. Als nächstes werden einige dieser Modelle vorgestellt.

- **Prozeß und Kanäle** (*task and channels*) – Dieses Modell nimmt an, daß ein paralleles Programm aus mehreren simultan ausgeführten Prozessen besteht und daß die Anzahl der Prozesse in der Zeit variieren kann. Ein Prozeß besteht wiederum aus einem sequentiellen Programm und einem lokalen Speicher, wobei eine Menge von *inports* und *outports* die Schnittstelle zu der „Außenwelt“ bildet. Neben dem Lesen vom lokalen Speicher bzw. dem Schreiben in den lokalen Speicher kann ein Prozeß noch vier weitere Grundoperationen durchführen: Nachrichten an seine *outports* senden, Nachrichten an seine *inports* empfangen, neue Prozesse erzeugen bzw. beenden. *Outport-Inport*-Paare können über die Kanäle verbunden werden. In Abbildung 2.3 sind Prozesse, Kanäle und Interaktionen zwischen Prozessen zu sehen.

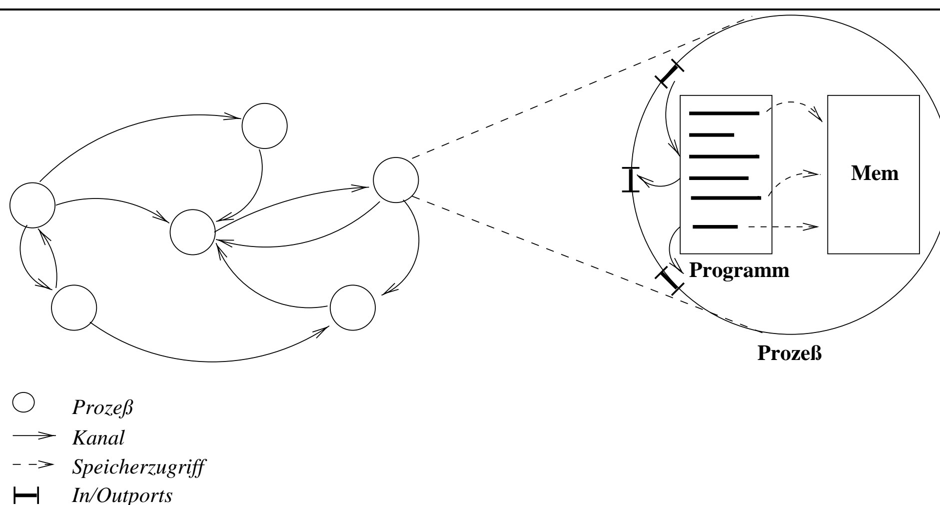


Abbildung 2.3: Prozeß-Kanäle-Programmiermodell

Der wesentliche Unterschied von diesem Programmiermodell zur Nachrichtenkopplung ist, daß die Kommunikationsstruktur des parallelen Programmes hier viel genauer spezifiziert werden muß. Anstatt die Kommunikation zwischen Prozessen in verschiedenen Prozessoren zu spezifizieren, muß die Kommunikation zwischen Kanälen spezifiziert werden, wobei ein Prozeß in der Regel über mehrere Kanäle verfügt.

- **Nachrichtenkopplung** (*message passing*) – Dies ist wahrscheinlich das derzeit am meisten benutzte parallele Programmiermodell [21]. Bei einem nachrichtengekoppelten Programm, wie auch bei einem Prozeß-Kanäle-Programm, kapselt jeder erzeugte Prozeß seine lokalen Daten. Jeder Prozeß wird durch einen Namen identifiziert. Prozesse interagieren durch das Senden und Empfangen von Nachrichten, wobei die Prozeßnamen dabei benutzt werden. Im Vergleich zu dem Prozeß-Kanäle-Modell wird bei dem Nachrichtenkopplung eine Nachricht an einen Prozeß geschickt, anstatt eine Nachricht an einen Kanal. Dieses Programmiermodell stellt die geringsten Anforderungen an eine Parallelrechnerarchitektur [54], da die einzelnen Prozessoren vollkommen unabhängig voneinander arbeiten und über eine Kommunikationsschnittstelle verfügen, die es ihnen gestattet, mit anderen Prozessoren Nachrichten auszutauschen. Die Frage nach einer Speicherkonsistenz stellt sich hier nicht, da jeder Prozeß nur seinen eigenen Speicher kennt und verwalten muß.

  Programmierumgebungen wie PVM (*Parallel Virtual Machine*) [15], MPI (*Message Passing Interface*) [8] oder P4 [6] sind typische Vertreter dieses parallelen Programmiermodells.

- **Datenparallelität** – Dieses parallele Programmiermodell nutzt die Parallelität aus, die auf der Ausführung der selben Operation synchron auf verschiedenen Elementen einer Datenstruktur basiert. Ein datenparalleles Programm besteht aus einer Menge solcher Operationen. Übersetzer für datenparallele Programme brauchen normalerweise vom Benutzer Information über die Datenverteilung auf die Prozessoren, d.h. der Benutzer muß festlegen, wie die Daten den Prozessoren zugeordnet werden.

  Unterschiede zu der Nachrichtenkopplung sind die synchrone Semantik und der globale Adreßraum, wobei hier ebenfalls explizite Kommunikationsoperationen verwendet werden.

  Typische Vertreter dieses Modells sind alle datenparallelen Programmiersprachen basierend auf Fortran oder C und deren Derivaten, wie z.B. HPF [32] und DPC(*Data Parallel C*) [26].

- **Gemeinsame Variablen** – Bei diesem Programmiermodell kommunizieren die verschiedenen Prozesse über einen gemeinsamen Adreßraum, woraus sie asynchron lesen bzw. in den sie asynchron schreiben. Die Speicherkonsistenz wird auf Architekturebene gewährleistet. Um den Zugriff auf den gemeinsamen Speicher zu verwalten, werden bestimmte Mechanismen benutzt, wie z.B. Schloßvariable (*locks*) und Semaphor. Dieses Modell vereinfacht die Entwicklung von Programmen, weil es keinen Bedarf an explizite Kommunikation zwischen Erzeuger und Verbraucher gibt.

  Die Vertreter dieses Modells sind Programmierumgebungen (Betriebssystemerweiterungen und sog. *Thread*-Bibliotheken) für Multiprozessorsysteme verschiedenster Hersteller.

In der vorliegenden Arbeit werden sowohl das Nachrichtenkopplung- als auch das Datenparallelitäts-Programmiermodell eingesetzt. Bei dem System, das die Einstellung des Parallelitätsgrades realisiert, wird die Nachrichtenkopplung benutzt. Bei den Benchmarks werden Nachrichtenkopplung und Datenparallelität eingesetzt.

2.4 Charakterisierung paralleler Programme

In diesem Abschnitt werden Kennzahlen zur Charakterisierung paralleler Programme vorgestellt. Auf Basis dieser Kennzahlen können parallele Algorithmen bewertet und miteinander verglichen werden.

Die Definitionen der drei nachfolgend vorgestellten Kennzahlen stammen aus [21] und dienen als Grundlage für die Definition komplexerer Kennzahlen und Leistungsmodelle. Auf sie wird vielmals in dieser Arbeit zurückgegriffen.

Berechnungszeit (T_{comp}, *computation time*) – Die Berechnungszeit eines Algorithmus ist die Zeit, die für Rechenoperationen verwendet wird. Die Berechnungszeit ist von der Problemgröße abhängig. Falls es bei einem parallelen Algorithmus replizierte Aktivitäten gibt, ist die Berechnungszeit auch von der Anzahl der Prozesse oder Prozessoren abhängig, wobei T_{comp} in diese Situation der maximalen Zeit unter der beteiligten Prozessoren entspricht. Bei einem Parallelrechner, der aus Prozessoren unterschiedlicher Art besteht (sog. *heterogener Parallelrechner*), kann die Berechnungszeit auf den verschiedenen Prozessoren variieren.

Die Berechnungszeit hängt auch mit den Charakteristika und den Speichersysteme der Prozessoren zusammen. Änderungen an der Problemgröße bzw. an der Anzahl der Prozessoren können die Leistung des *Cache*-Speichers oder die Effektivität des Prozessorpipelining beeinträchtigen. Folge daraus ist, daß die totale Berechnungszeit eines Algorithmus, d.h. die über alle beteiligten Prozessoren kumulative Berechnungszeit, nicht unbedingt konstant bleibt, wenn sich die Anzahl der benutzten Prozessoren ändert.

Kommunikationszeit (T_{comm}, *communication time*) – Die Kommunikationszeit eines Algorithmus ist die Zeit, in der Nachrichten gesendet bzw. empfangen werden. Zwei Arten der Kommunikation können unterschieden werden: Die Intra-Prozessor- und die Inter-Prozessor-Kommunikation. Bei der ersten befinden sich die zwei Prozesse, die miteinander kommunizieren möchten, auf demselben Prozessor, wobei bei der zweiter Art der Kommunikation die Prozesse auf unterschiedlichen Prozessoren sind.[1]

Die Kosten, die bei einer Inter-Prozessor-Kommunikation zwischen zwei Prozessen zustande kommen, können durch zwei Parameter repräsentiert werden: Die Startzeit t_s (*startup time*) und die Übertragungszeit pro Wort t_w (*transfer time per word*). Die erste ist die Zeit, die für die Initialisierung der Kommunikation benötigt wird, während t_w die Bandbreite des Kommunikationsmediums ist, das die zu kommunizierenden Prozessoren verbindet. Die Übertragungszeit einer Nachricht, die aus L Worten besteht, beträgt:

$$T_{msg} = t_s + t_w \times L$$

Interessant ist, daß für einen sehr großen L nur t_w von Bedeutung ist, wobei für sehr kleine Nachrichten t_s dagegen eine dominierende Rolle spielt.

[1] Wenn nur ein Prozeß pro Prozessor vorgesehen ist, handelt es sich immer um eine Inter-Prozessor-Kommunikation.

Bereitschaftszeit (T_{idle}, *idle time*) – Die Bereitschaftszeit ist die Zeit, während der ein Prozeß zwar bereit ist, aber nichts zu berechnen hat oder während die für die Berechnung benötigten Daten nicht verfügbar sind. Die erste Situation kann durch Techniken zur Lastbalancierung [47] vermieden werden. Bei der zweiten Situation befindet sich der Prozeß im Bereitschaftsmodus, weil die benötigten Daten für die Fortsetzung der Berechnung gerade geholt werden. Diese Bereitschaftszeit kann aber ausgenutzt werden, in dem die betroffenen Prozessoren andere Prozesse ausführen. Diese Technik wird Überlappung (*overlapping*) der Berechnung und Kommunikation gennant und ist in Abbildung 2.4 dargestellt. Die durchgehenden Linien repräsentieren Berechnungen, gestrichelte Linien Kommunikationsoperationen, sowie P_1 und P_2 Prozessoren. Sowohl in Abbildung 2.4(a) als auch in 2.4(b) erzeugt Prozessor P_1 eine Anforderung an Prozessor P_2 im Zeitpukt $t + 2$ und empfängt eine Antwort im Zeitpukt $t + 8$. In (a) hat P_1 nichts mehr zu berechnen bis P_2 ihm eine Nachricht schickt, in (b) dagegen wechselt P_1 vom Prozeß A zu Prozeß B bis P_2 ihm eine Nachricht geschickt hat. Diese Technik ist aber nur dann effektiv, wenn die Kosten, die mit dem Kontextwechsel zwischen Prozessen gebunden sind, nicht höher als die zu vermeidenden Bereitschaftszeit sind.

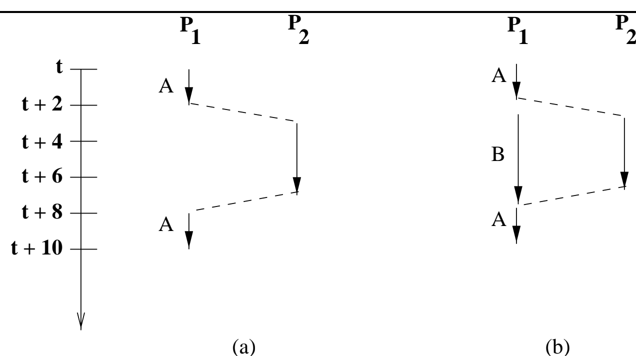


Abbildung 2.4: Überlappung von Berechnung und Kommunikationsoperation

Ausführungszeit (T, *execution time*) – Die Ausführungszeit eines parallelen Programmes mit p Prozessoren setzt sich aus den drei bereits vorgestellten Kennzahlen zusammen, wobei sie auf zwei verschiedene Arten definiert werden kann. Bei der ersten Art ist die Ausführungszeit T die Summe der Berechnungs-, der Kommunikations- und der Bereitschaftszeit an einem Prozessor j:

$$T = T^j_{comp} + T^j_{comm} + T^j_{idle}$$

Da die für einen Prozessor individuellen Zeitanteile oft nicht gemessen werden können, ist es sinnvoller, die Ausführungszeit in einer anderen Weise darzustellen. Hier wird die Ausführungszeit T als eine Summe verschiedener Zeitanteile über alle Prozessoren gebildet und diese durch die Anzahl der Prozessoren p dividiert:

$$T = \frac{1}{p}\left(\sum_{i=0}^{p-1} T^i_{comp} + \sum_{i=0}^{p-1} T^i_{comm} + \sum_{i=0}^{p-1} T^i_{idle}\right)$$

Parallelitätsgrad – Der Parallelitätsgrad eines Programmes gibt an, wieviel Parallelismus das Programm zu einem bestimmten Zeitpunkt anzubieten hat. In der vorliegenden Arbeit wird Parallelitätsgrad letztendlich als Synonym für die Anzahl von Prozessoren angewendet, die vom betrachteten Programm zu einem bestimmten Zeitpunkt in Anspruch genommen werden können.

Parallelitätsprofil – Das Parallelitätsprofil eines Programmes ist der Verlauf des Parallelitätsgrades p_t über die Ausführungszeit t des Programmes. In Abbildung 2.5 ist zu sehen, daß der Parallelitätsgrad eines Programmes in der Zeit variiert, d.h. es gibt Phasen, in denen das Programm mehr Parallelismus anzubieten hat (z.B. bei p_{max}), und andere Phasen, in denen weniger und/oder feinkörnigere parallele Aktivitäten vorhanden sind (z.B. bei p_{min}). Das hat Kumar in [33] im Jahr 1988 gezeigt, wobei er ein Werkzeug mit dem Ziel entwickelt hat, den Parallelismus für FORTRAN-Programme zu messen. Der Parallelitätsprofil kann auch als die Anzahl der von einem Programm benutzten Prozessoren interpretiert werden [28], wenn beliebig viele Prozessoren zur Verfügung stehen.

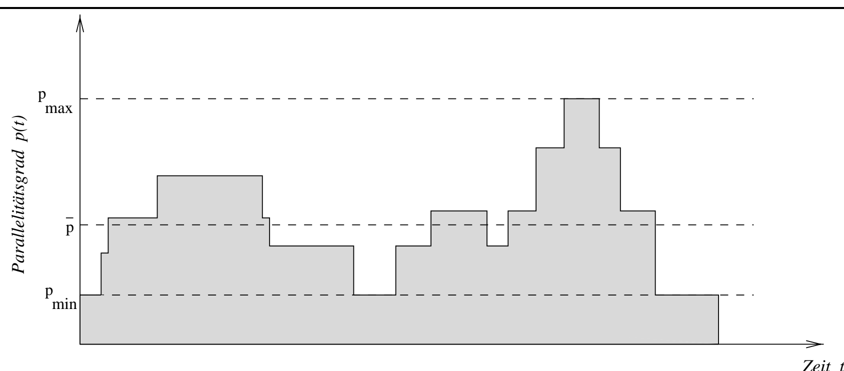


Abbildung 2.5: Parallelitätsprofil eines Programmes

Beschleunigung (S_p, *speed-up*) – Die Beschleunigung stellt das Verhältnis zwischen der Ausführungszeit eines Programmes auf einem Prozessor und der auf p Prozessoren dar:

$$S_p = \frac{T_1}{T_p}$$

Die erzielte Beschleunigung sagt letztendlich, ob und wieviel sich die Ausführungszeit des Programmes reduziert bzw. vergrößert hat, wenn p Prozessoren statt einem Prozessor eingesetzt werden. Bei einer Beschleunigung von p auf p Prozessoren spricht man von einer *linearen Beschleunigung*, die wegen der Konkurrenz um gemeinsame Betriebsmittel, den zusätzlichen Kosten, die durch die Kommunikation zwischen Prozessoren entstehen, und der begrenzten Anzahl paralleler Aktivitäten, die das Programm anzubieten hat, nur schwer zu erreichen ist. Im Normalfall ist die Beschleunigung nicht linear und wächst auf Kosten der Effizienz.

Effizienz (E_p, *efficiency*) – Die Effizienz stellt das Verhältnis zwischen der Beschleunigung, die mit dem Einsatz von p Prozessoren erreicht wurde, und der Prozessoranzahl dar:

$$E_p = \frac{S_p}{p}$$

Die erzielte Effizienz sagt, wie gut die p Prozessoren ausgelastet sind bzw. wie gut ein Algorithmus die Betriebsmittel eines Parallelrechners ausnutzen kann. Wenn nur ein Prozessor benutzt wird, beträgt die Effizienz 1. In dem Fall, daß die Effizienz bei 1 bleibt, wenn mehrere Prozessoren dazu kommen, ist die Beschleunigung linear.

Sowohl die Beschleunigung als auch die Effizienz können *relativ* oder *absolut* sein [21]. Sie sind relativ, wenn für ihre Erfassung die Ausführung des parallelen Algorithmus auf einem Prozessor berücksichtigt wird. Absolut sind sie, wenn der bestbekannte sequentielle Algorithmus eingesetzt wird.

Wie in [13] beschrieben, können die Beschleunigung und die Effizienz in der Praxis nicht gleichzeitig wachsen. Die Beschleunigung wird normalerweise auf Kosten der Effizienz gesteigert, da die gesamte Bereitschaftszeit mit der Anzahl der Prozessoren wächst. Das passiert wegen der Konkurrenz, der Kommunikation und der Programmstruktur, wie schon oben erläutert wurde.

Kosten-Nutzen-Relation (*cost-benefit relation*) – Wenn man die Kurven T_p (Ausführungszeit) und E_p (Effizienz) in Abbildung 2.6 etwa im Bereich des Minimums der Ausführungszeitkurve betrachtet, stellt man fest, daß bei der Erhöhung des Parallelitätsgrades um 1 der Gewinn bezüglich der Ausführungszeit praktisch 0 ist, die Effizienz sich jedoch kontinuierlich verringert. Bei einem sehr geringen Parallelitätsgrad kann dagegen die Erhöhung um 1 eine erhebliche Reduktion der Ausführungszeit bewirken, während sich die Effizienz nicht in demselben Maß verschlechtert. Diese Beobachtung führt zu der Idee, die Kosten-Nutzen-Relation zu minimieren [28]. Diese Relation entsteht durch die Bildung des Quotienten aus den Kosten $C_p$und dem Nutzen S_p. Äquivalent dazu kann man auch die reziproke Nutzen-Kosten-Relation maximieren.

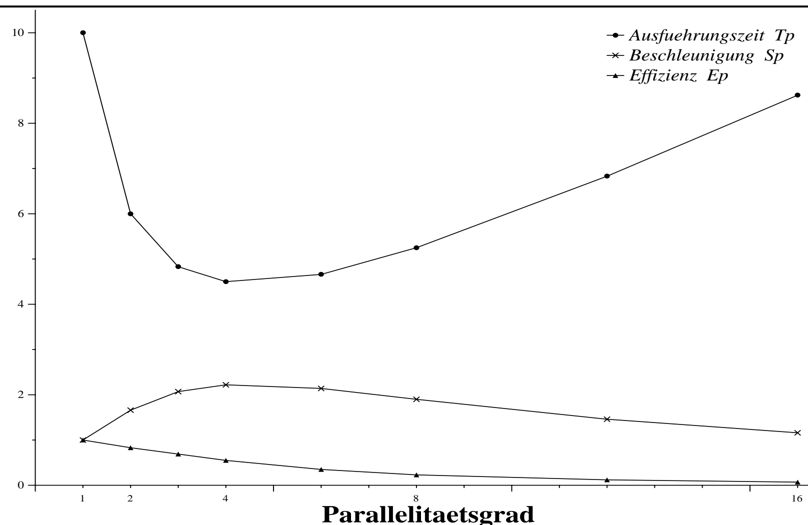


Abbildung 2.6: Zusammenhang zwischen S_p, E_p und T_p

Die Kosten sind durch

$$C_p = T_p \times p$$

repräsentiert, wobei p die Anzahl der benutzten Prozessoren und T_p die Ausführungszeit des Programms auf p Prozessoren darstellt. Die Kosten resultieren aus dem Einsatz mehrerer Prozessoren, anstatt nur einem, für die Ausführung eines Programmes.

Der Nutzen S_p ist die Verbesserung oder Verschlechterung der Ausführungszeit durch die Benutzung mehrerer Prozessoren im Vergleich zu dem Einsatz von nur einem Prozessor, was letztendlich der Beschleunigung eines Programmes entspricht.

Die Kosten-Nutzen-Relation R_{CB} ist damit:

$$R_{CB} = \frac{T_p^2 \times p}{T_1}$$

Die Kennzahl Kosten-Nutzen-Relation beantwortet die Frage, inwiefern es sich lohnt, zusätzliche Prozessoren einem parallelen Programm zuzuordnen.

2.5 Modellierung der Leistung paralleler Programme

Hier werden drei Techniken präsentiert, die für die Charakterisierung der Leistung paralleler Programme benutzt werden.

2.5.1 Das Amdahl'sche Gesetz

Nach dem Amdahl'schen Gesetz [1] besteht ein paralleler Algorithmus aus einem parallelen (α) und einem sequentiellen (β) Teil. Die Beschleunigung (siehe Abschnitt 2.4), die durch den Algorithmus erzielt werden kann, ist wegen des sequentiellen Teils begrenzt. Die maximale Beschleunigung, die auf einem Parallelrechner erzielt werden kann, ist $1/(1-\alpha)$, wenn es keine Grenze für die Anzahl der Prozessoren gibt und $T_1 = 1$. Dise Erkenntnis wird im folgenden gezeigt.

Die Ausführungszeit T_p auf p Prozessoren setzt sich aus der Ausführungszeit des sequentiellen und des parallelen Teils zusammen:

$$T_p = (1-\alpha) + \frac{\alpha}{p} \tag{2.1}$$

Wie im Abschnitt 2.4 erläutert, wird die Beschleunigung eines Programmes durch die folgende Formel repräsentiert:

$$S_p = \frac{T_1}{T_p} \tag{2.2}$$

Wenn die Prozessoranzahl unendlich wachsen kann, folgt aus (2.1) und (2.2):

$$S_p = \frac{1}{(1-\alpha) + \frac{\alpha}{p}} = \frac{1}{(1-\alpha)}$$

Wenn der sequentielle Teil eines Programmes z.B. 5% beträgt, beträgt die maximale Beschleunigung, die auf einem Parallelrechner erzielt werden kann, 20.

2.5.2 Asymptotische Analyse

Die Leistung eines parallelen Algorithmus kann auch durch eine asymptotische Analyse modelliert werden. Das Problem dabei ist, daß eine asymptotische Analyse Faktoren, die in vielen Fällen wichtig sein können, unberücksichtigt läßt. Die wirklichen Kosten eines Algorithmus für eine Problemgröße von N können z.B. $10N + NlogN$ betragen, wobei eine asymptotische Analyse die Kosten als $NlogN$ modellieren würde. Dies wäre für mittelgroße bzw. große Problemgrößen realistisch, für kleine Problemgrößen aber nicht angebracht. Ein zweiter Nachteil ist, daß asymptotische Analyse auf idealisierten Maschinenmodellen basiert, die weit entfernt von realistischen Maschinen liegen können, wobei Aspekte wie Speicherhierarchie unberücksichtigt bleiben.

Eine asymptotische Analyse von parallelen Algorithmen basiert auf dem PRAM- und auf dem LogP-Modell, die im folgenden beschrieben werden.

Das PRAM-Modell – PRAM steht für *Parallel Random Access Machine* [20] und modelliert einen idealisierten speichergekoppelten Parallelrechner, der aus p identischen Prozessoren besteht. Die Prozessoren arbeiten synchron, greifen auf einem gemeinsamen Speicher zu und kommunizieren miteinander, ohne daß Kosten dabei entstehen. Außerdem enthalten alle Programmspeicher das gleiche Programm, führen dieselbe oder aber auch verschiedene Rechenoperationen gleichzeitig aus.

Das LogP-Modell – Hier wird ein nachrichtgekoppelter Parallelrechner, bei dem die Prozessoren unabhängig voneinander, vollkommen asynchron arbeiten, modelliert. Die Prozessoren kommunizieren über ein Netzwerk miteinander. Um die Kommunikation zu berücksichtigen, definiert das LogP-Modell [10] vier Kennzahlen, woraus der Name des Modells auch entsteht. Die Kennzahlen sind:

- **Latenzzeit** (L, *latency*) – obere Schranke der Verzögerungszeit einer Kommunikationsoperation beim Austausch einer kleinen Nachricht (d.h. wenige Worte) zwischen zwei Knoten.

- **Aufwand** (o, *overhead*) – Zeitbedarf für den Sende- bzw. Empfangsvorgang. Während dieser Zeit kann der sendende Prozessor keine weiteren Operationen ausführen.

- **Abstand** (g, *gap*) – untere Schranke für die Zeit, die zwischen aufeinanderfolgenden Sende- bzw. Empfangsoperationen eingehalten werden muß.

- **Prozessoranzahl** (P, *processors*) – Anzahl der Prozessoren, wobei jeder Prozessor über einen lokalen Speicher verfügen kann.

Das Modell hat einige Einschränkungen: Zu einem bestimmten Zeitpunkt sind maximal $\lceil L/g \rceil P$ Nachrichten im Netz. Außerdem werden keine Konflikte im Verbindungsnetz berücksichtigt. Eine dritte Einschränkung ist die Tatsache, daß größere Nachrichten, bei denen die Latenzzeit von der Nachrichtenlänge abhängt, nicht beachtet werden.

Die asymptotische Analyse ist sicherlich relevant für die Entwicklung paralleler Programme. Die Ergebnisse einer asymptotischen Analyse müssen aber sorgfältig untersucht werden, indem die Rechnerarchitektur, für die die Ergebnisse erzielt wurden, die Koeffiziente, die angewendet werden müssen, die Problemgröße und die Prozessoranzahl klargestellt werden.

2.5.3 Planbare Algorithmen

W. Zimmermann [58] und W. Löwe [36] haben sich mit der Transformation paralleler Programme beschäftigt, die nach dem PRAM-Modell entwickelt wurden. Diese parallelen Programme sollen in Programme transformiert werden, die das oben beschriebenen LogP-Modell einsetzen. Das Ziel dabei ist, von einem theoretischen, abstrakten Programmiermodell zu einem Modell von praktischer Relevanz überzugehen, wobei die Ausführungszeiten der übersetzten Algorithmen auf der LogP-Maschine optimiert werden sollen. Die Transformationen beruhen auf einer Subklasse von PRAM-Programmen[2], die durch einen planbaren Algorithmus gegeben werden können. Planbare Algorithmen sind Algorithmen, die sich durch ein eingabeunabhängiges Kommunikationsverhalten charakterisieren lassen, wobei das Verhalten nur vom Umfang der Eingabe abhängt, nicht jedoch von deren Inhalt[3].

Diese Methode ermöglicht es, die maximale Ausführungszeit eines PRAM-Programmes auf verfügbaren Rechnern statisch zu berechnen. Das LogP-Modell ist ein geeigneter Kompromiß zwischen der bei der parallelen Programmierung erzielten Einfachheit und der Realitätsnähe bzw. praktischen Relevanz. Die o.g. Einschränkungen des LogP-Modells bleiben aber erhalten.

2.5.4 Methode nach Foster

Ian Foster definiert in [21] eine neue Methode zur Modellierung der Leistung paralleler Algorithmen, die in der vorliegenden Arbeit bei der Validierung des mathematischen Modells, das den optimalen Parallelitätsgrades eines Programmes berechnet, benutzt wird. Foster behauptet, daß eine gute Leistungsmodellierung zwei Kriterien erfüllen muß: Erstens muß sie fähig sein, Beobachtungen bzw. Phänomene zu erklären, und zweitens muß sie künftig auftretende Situationen vorhersagen können, wobei von unwichtigen Details abstrahiert werden soll. Da sowohl das Amdahl'sche Gesetz als auch asymptotische Analyse das erste Kriterium nicht erfüllen, ist man auf der Suche nach einer vollständigeren Methode zur Modellierung der Leistung paralleler Algorithmen, die aber unbrauchbare Details nicht berücksichtigt.

Nach Foster wird die Ausführungszeit T eines Programmes in Abhängigkeit von der Problemgröße N, der Anzahl von Prozessoren p, der Anzahl von Prozessen U und anderer Charakteristika des Algorithmus sowie der Hardware definiert:

$$T = f(N, p, U, ...)$$

Das Ziel dabei ist, Formeln zu entwickeln, die die Ausführungszeit von N, p, usw. abhängend spezifizieren. Bei der Ermittlung der Ausführungszeit T werden die Berechnungs-, die Kommunikations- sowie die Bereitschaftszeit wie bei der im Abschnitt 2.4 definierten Charakterisierung paralleler Programme betrachtet. Nachteil dieser Methode ist, daß sie Aspekte wie Speicherhierarchie bei der Modellierung der Leistung paralleler Algorithmen unberücksichtigt läßt.

[2] Aus praktischer Sicht besteht diese Subklasse aus einer großen Menge von PRAM-Programmen, z.B. FFT, Finite-Elemente-Methode, PDE-Löser, Matrix Multiplikation, usw.

[3] Um PRAM-Programme in LogP-Programme zu transformieren, sind nach Zimmermann und Löwe drei Schritte erforderlich: Erstens muß die Anzahl der eingesetzten Prozessoren auf die Anzahl der zur Verfügung stehenden Prozessoren reduziert werden, zweitens muß das synchrone auf ein äquivalentes asynchrones Programm umgeschrieben werden. Drittens soll von einem verteilten statt einem gemeinsamen Speicher ausgegangen werden.

2.6 Entwurf paralleler Software

Für den Entwurf paralleler Software kann nicht ein einfaches Rezept vorgegeben werden, weil dabei viel an Kreativität gefordert ist. Dieser Prozeß kann aber durch eine spezifische Vorgehensweise unterstützt werden, indem der Weg von der Problemspezifikation bis hin zum parallelen Algorithmus beschrieben wird.

Foster [21] und Ungerer [52] strukturieren den Entwurfsprozeß paralleler Software in vier Phasen:

- **Partitionierung** (*partitioning*) – Die Berechnungsschritte und die Daten, auf denen die Berechnungen ausgeführt werden, werden in kleine Aufgaben aufgeteilt. Praktische Erwägungen wie die Maschinenkonfiguration, die Anzahl der Prozessoren und die Struktur des Verbindungsnetzwerkes werden in dieser Phase ignoriert. Das Hauptziel ist, möglichst viel Parallelität zu entfalten.

- **Kommunikation** (*communication*) – In dieser Phase werden Kommunikationsanforderungen festgelegt, und geeignete Kommunikationsstrukturen und Algorithmen definiert.

- **Bündelung** (*agglomeration*) – Die Aufgaben und Kommunikationsstrukturen, die in den ersten zwei Phasen festgelegt wurden, werden bzgl. der Leistungsanforderungen und bzgl. Implementierungskosten evaluiert. Falls es notwendig ist, werden Aufgaben zu umfassenderen Aufgabenbereichen gebündelt, um die Leistung zu erhöhen oder die Entwicklungskosten zu sparen.

- **Abbildung** (*mapping*) – Die Aufgaben werden den Prozessoren so zugeordnet, daß die konkurrierenden Ziele der maximalen Prozessorauslastung und der minimalen Kommunikationskosten miteinander in Einklang gebracht werden. Die Prozesorzuteilung kann statisch vom Übersetzer festgelegt oder dynamisch erst zur Laufzeit durch Anwendung von Lastbalancierungsalgorithmen vorgenommen werden.

Die verschiedenen Phasen sind in Abbildung 2.7 gezeigt, wobei das Ergebnis jeder Phase graphisch dargestellt ist. Die ersten zwei Phasen konzentrieren sich auf die Entfaltung von Parallelität sowie auf eine gute Skalierbarkeit und suchen einen Algorithmus mit diesen Eigenschaften. Die dritte und vierte Phasen behandeln die Lokalität und weitere leistungsorientierte Eigenschaften.

Die beschriebene Vorgehensweise bleibt für alle Zielprogrammstrukturen gleich. Beispielsweise wird bei einer SPMD(*Single Programm Multiple Data*)-Programmstruktur jedem Prozessor dieselbe Aufgabe zugeordnet.

Um die vorliegende Arbeit in den Entwurfsprozeß paralleler Software einzuordnen, wird von Fosters Phasenaufteilung Gebrauch gemacht. Die automatische Einstellung des Parallelitätsgrades ist Teil der Abbildungsphase, wobei hier eine dynamische und *iterative Abbildung* angewendet wird. Die Abbildung wird deswegen iterativ genannt, weil sie zur Laufzeit wiederholt durchgeführt wird, und zwar jedesmal, wenn sich der Parallelitätsgrad ändert.

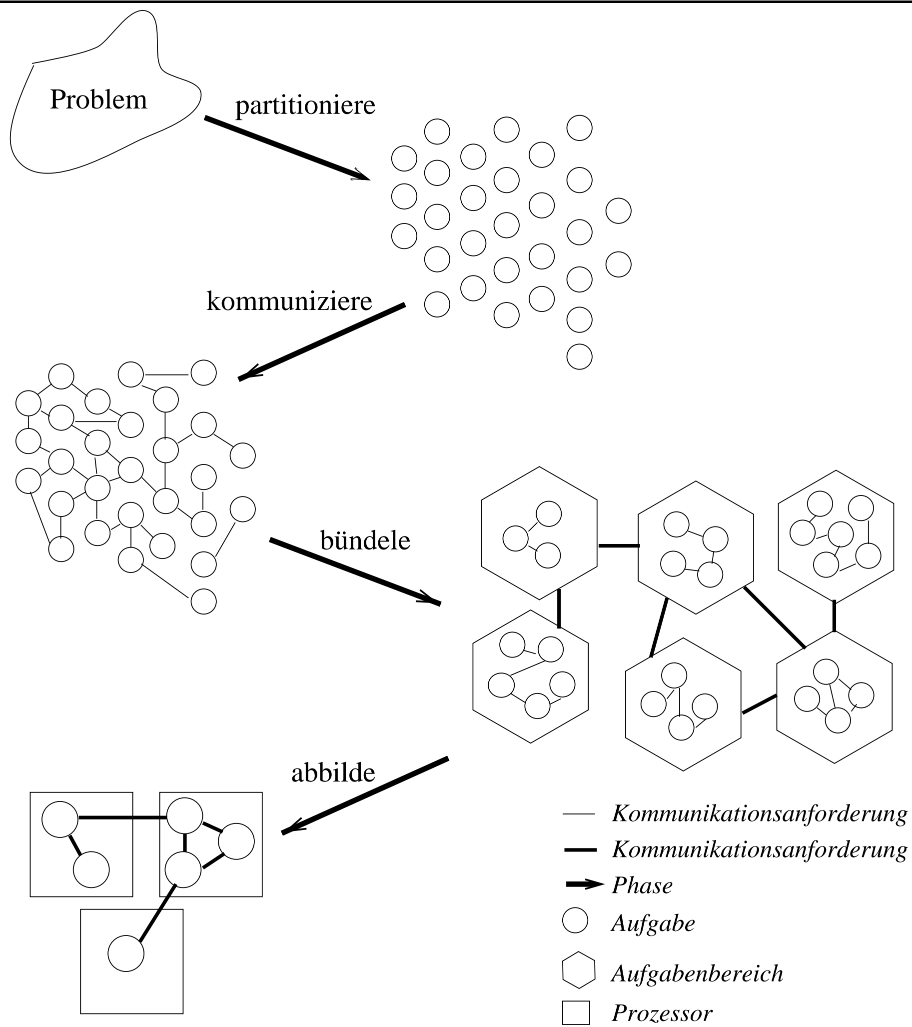


Abbildung 2.7: Phasen des Entwurfsprozesses paralleler Programme

Kapitel 3

Stand der Forschung

Im folgenden wird der Stand der Forschung vorgestellt. Es werden Arbeiten aus drei Gebieten präsentiert und mit der vorliegenden Arbeit verglichen. Zuerst werden Beiträge zur Verbesserung der Auslastung von Parallelrechnern erläutert, danach Arbeiten, die durch die Bestimmung des Parallelitätsgrades zur Verbesserung der Leistung paralleler Programme beitragen. Als letztes werden diejenigen Beiträge vorgestellt, die eine Kombination von den zwei erstgenannten Gebieten berücksichtigen. Die vorliegende Arbeit ist dem letzten Gebiet zuzuordnen.

3.1 Verbesserung der Auslastung von Parallelrechnern

In diese Kategorie fallen alle Arbeiten, die sich mit Prozessorzuteilung beschäftigen und als Ziel eine Verbesserung der Auslastung des Gesamtsystems haben. Die Verwandschaft zu der voliegenden Arbeit besteht darin, daß auch die vorliegende Arbeit eine Verbesserung der Auslastung von Parallelrechnern als eines der Ziele hat. Trotz dieser Verwandschaft unterscheiden sich die Arbeiten im Lösungsansatz. Die vorliegende Arbeit befaßt sich nicht mit der Prozessorzuteilung in Parallelrechnern. Sie geht davon aus, daß die Prozessorzuteilung schon gelöst ist, aber daß der Bedarf an Prozessoren für verschiedene parallele Programme und ihre unterschiedliche Phasen (siehe Abbildung 2.5) nicht genau berücksichtigt, bzw. nicht überprüft wird. Außerdem betrachtet die vorliegende Arbeit die Einstellung des Parallelitätsgrades auf Programmebene anstatt auf Systemebene.

H.-U. Heiss an der Universität Karlsruhe klassifiziert in [28] die Prozessoraufteilung in statische, semi-dynamische und dynamische Aufteilung. Bei der *statischen Aufteilung* ist eine Menge paralleler Programme und die Funktionen vorgegeben, die die Beschleunigung der Programme wiedergeben. Auf Basis dieser Funktionen werden die Parallelitätsgrade der Programme berechnet und die Prozessoren für die gesamte Abarbeitungszeit fest aufgeteilt. Bei der *semi-dynamischen Aufteilung* ist eine dynamische Menge von Programmen gegeben, die jeweils einen konstanten Bedarf an Prozessoren haben (sog. statische Programme). Auch hier wird davon ausgegangen, daß der optimale Parallelitätsgrad eines jeden Programmes bekannt ist. Die *dynamische Aufteilung* erlaubt dagegen, daß Programme während ihrer Abarbeitung weitere Prozessoren allozieren bzw. freigeben. Das Verfahren setzt aber voraus, daß die Beschleunigungsfunktionen der Programme zur Verfügung stehen. Die Idee ist, demjenigen Programm inkrementell einen weiteren Prozessor zuzuteilen, dessen Beschleunigung dadurch am größten wird [43]. Damit wird schrittweise sichergestellt, daß im Hinblick auf den Durchsatz vom Betriebsmittel *Prozessor* bestmöglicher Gebrauch gemacht wird. Keine Implementierung der Technik in der letzten Kategorie ist bekannt.

Im Vergleich zur dynamischen Aufteilung wird der Parallelitätsgrad bei der vorliegenden Arbeit seitens des Programms, statt seitens des Betriebssystems, eingestellt. Außerdem wird der Parallelitätsgrad nicht nur nach der Beschleunigung eingestellt, sondern auch alternativ nach der Kosten-Nutzen-Relation und Effizienz.

M.S. Squillante, F. Wang und M. Papaefthymiou an dem Forschungszentrum T.J.Watson bzw. an der Yale Universität präsentieren in [50, 53, 49] die Technik *gang scheduling* (zuerst von Ousterhout [42] präsentiert), die die Betriebsmittel auf verteilten Systemen verwaltet. *Gang scheduling* kombiniert zwei Techniken: Erstens die Zeitanteil-Technik oder auch *time sharing* genannt, wodurch alle Prozessoren einem einzigen Programm für eine vorgegebene Zeit zugeordnet werden. Die zweite Technik ist die sog. Platzanteil-Technik oder auch *space sharing* genannt, bei der Programme mit der gleichen Anforderung an Prozessoren zu derselben Gruppe (*gang*) gehören. Jeder Gruppe wird eine Prozessorunterteilung (*partition*) für eine gegebene Zeitdauer zugeordnet. Wenn diese Zeit vorbei ist, geschieht ein Kontextwechsel für die Ausführung der nächsten Programmgruppe. Hier wird die Anzahl der den Programmen zugeteilten Prozessoren heuristisch festgelegt.

A. Gupta, A. Tucker und S. Urushibara beschreiben in [23] eine Studie, bei der die Leistung verschiedener Prozessozuteilungsstrategien durch Simulation untersucht wurde. *Gang scheduling* und *process-control*-Techniken (d.h. Techniken, die erlauben, daß Programme eine in der Zeit variierende Anzahl von Prozessen abhängig von der Anzahl der freien Prozessoren haben) zeigten die beste Leistung, wobei die Systemauslastung bei 71% lag. Kommerzielle Systeme wie CM-5 [34] und Intel Paragon [9] benutzen eine einfache *gang-scheduling*-Technik, die eine vorgegebene Prozessorunterteilung anwendet. Es gibt auch Variationen der *gang-scheduling*-Technik, die z.B. flexible Prozessorunterteilungen bzw. flexible Zeitanteile erlauben.

Bei der vorliegenden Arbeit wird der Parallelitätsgrad auf Basis von Messungen eingestellt, die zur Laufzeit stattfinden, anstatt ihn basierend auf Heuristiken statisch festzulegen. Außerdem ist bei der Arbeit von M. Squilante et al. eine Datenumverteilung trotz der Zielarchitektur, die über verteilten Speicher verfügt, nicht notwendig, weil sich der Parallelitätsgrad der Gruppe bis zum Ende der Ausführung der Programme nicht mehr ändert. Wie in Abschnitt 2.4 erwähnt, variiert der Parallelitätsgrad eines Programmes in der Zeit, was beim Einsatz dieser Methode zu einer Verschlechterung der Leistung von parallelen Programmen führen kann.

Die drei nächsten Arbeiten beschäftigen sich mit der dynamischen Zuordnung von Prozessoren zu Prozessen in Architekturen mit gemeinsamem Speicher und haben eine gute Auslastung des gesamten Systems als Ziel. Diese Arbeiten, die sich auf Architekturen der o.g. Art beschränken, berücksichtigen keine Datenumverteilung. Daraus folgt, daß diese Ansätze auf eine Architektur, die über verteilten Speicher verfügt, nicht anwendbar sind.

Der Prototyp **Minos von C. McCann, R. Vaswani und J. Zahorjan an der Universität Washington** [39] schlägt eine Strategie für die Prozessorzuordnung in Mehrprozessorsystemen mit gemeinsamem Speicher vor. Die Strategie ist dynamisch, weil sie die Zuordnung von Prozessoren zu Prozessen je nachdem ändert, wieviel Parallelismus jeder Prozeß während der Ausführungszeit anzubieten hat. Ein Prozeß enthält eine variierende Menge von Threads und signalisiert dem Scheduler-Thread, wieviele dieser Threads laufbereit sind. Anhand dieser Information wird die Anzahl der zu den Prozessen zugeordneten Prozessoren geändert, nämlich sie wird größer oder kleiner je nachdem, ob der Prozeß gerade zu viele oder zu wenig Prozessoren belegt. Der Prototyp ist in C++ geschrieben und läuft auf einer Sequent-Symmetry-Maschine mit 20 Prozessoren unter dem Betriebssystem DYNIX. Die Leistung der dynamischen Prozessorzuordnung wurde mit anderen Strategien

(Round-Robin Job [35], Equipartition [51]) verglichen und hat die besten Ergebnisse geliefert. *Minos* beschäftigt sich aber nicht mit der Frage, wie die Anzahl der Threads pro Prozeß bestimmt wird. Es wird einfach angenommen, daß jeder Prozeß dem Scheduler-Thread die richtige Anzahl von Threads mitteilt, ohne daß diese Information bei der Ausführungszeit überprüft wird.

Wesentliche Unterschiede zu der vorliegenden Arbeit sind die abweichende Zielarchitektur und die Art der Bestimmung des optimalen Parallelitätsgrades, die bei der vorliegenden Arbeit dynamisch und basierend auf realen Messungen stattfindet.

Die Technik **LLPC (***Loop-Level Process Control***) von K. Yue und D. Lilja an der Universität Minnesota** [56] unterstützt die Anpassung der Anzahl von Prozessoren, die eine Anwendung für die Ausführung der einzelnen parallelen Abschnitte benutzen darf. Diese Anpassung basiert auf der aktuellen Systemlast. Die Realisierung dieser Technik wurde durch die Einstellung der Anzahl der Threads erzielt, die jeder parallelen Schleife dynamisch zugeordnet wird. Die Kriterien für die Anpassung hängen aber nur von zwei Größen ab, nämlich der Systemlast und der Granularität der Schleife, die nur statisch abgeschätzt wird. Wenn die Systemlast hoch ist, reduziert LLPC die Kontextwechselrate, indem den Anwendungen nur eine geringe Anzahl von Prozessoren zugeordnet werden, statt ihnen die statisch bestimmte maximale Anzahl der Prozessoren zu erlauben. Die Messungen wurden auf einem Silicon-Graphics-Challenge-System unter dem Betriebssystem IRIX durchgeführt, wobei alle Beispielprogramme durch den PFA (*Power Fortran Accelerator*) parallelisiert wurden. Die Ergebnisse zeigen eine Verbesserung im Vergleich zu der Technik, die für die Prozessorzuordnung durch Silicon Graphics zur Verfügung steht (*gang scheduling* [50, 16]), und zu der Technik *space sharing*. Bei der Benchmark-Sammlung „Perfect Club“ [4] konnte LLPC z.B. von 71% bis 85% der maximalen Beschleunigung in einer *standalone*-Ausführung der Programme erreichen.[1] Die Technik *space sharing* konnte dagegen nur gerade 56% der maximalen Beschleunigung erzielen. In einem anderen Testfall konnte LLPC von 47% bis 93% der maximalen Beschleunigung erreichen, während dieser Wert beim *gang scheduling* nur im Bereich von 28% bis 68% lag.

Neben der Zielarchitektur ist ein anderer Unterschied zu der vorliegenden Arbeit zu beobachten, nämlich daß die Einstellung des Parallelitätsgrades hier nur von der Systemlast und von einer statischen Abschätzung der Arbeit in der parallelen Schleife abhängt. Bei der vorliegenden Arbeit werden diese Parameter dynamisch kontrolliert, was die Methode auch auf irregulären Parallelismus anwendbar macht.

Die Arbeit von **A. Tucker und A. Gupta an der Universität Stanford** [51] schlägt einen Lösungsansatz für ein sehr spezifisches Problem vor, nämlich die Verschlechterung der Leistung von Programmen auf Multiprozessoren mit gemeinsamem Speicher, wenn die Anzahl der laufenden Prozesse viel größer ist als die Anzahl der zur Verfügung stehenden Prozessoren. Es handelt sich hier um eine Technik, die die Anzahl der laufenden Prozesse dynamisch kontrolliert, wobei die optimale Anzahl der Prozesse pro Programm durch eine zentrale Diensteinheit gegeben wird. Anhand dieser Information initialisieren bzw. beenden die Programme dynamisch Prozesse, um diese optimale Anzahl zu treffen. Diese Technik nimmt an, daß die Leistung paralleler Programme am besten ist, wenn die Gesamtanzahl der benutzten Prozesse gleich der Anzahl der zur Verfügung stehenden Prozessoren ist, wobei mehrere Programme gleichzeitig ausgeführt werden können. Im System können sowohl kontrollierte als auch nicht-kontrollierte Programme aktiv sein. Kontrollierte Programme

[1] Die *standalone*-Ausführung eines Programmes bezieht sich auf die Ausführung auf einem Rechner, auf dem kein weiteres Programm gleichzeitig läuft.

sind diejenigen, die unter der beschriebenen Technik laufen. Die Methode wurde auf einer Encore Multimax mit 16 Prozessoren implementiert. Die zentrale Diensteinheit kontrolliert regelmäßig die Gesamtanzahl der Prozesse im System, überprüft die Anzahl der Prozesse, die zu nicht kontrollierten Programmen gehören, subtrahiert diese von der gesamten Prozessoranzahl, um die Anzahl der Prozessoren festzulegen, die den kontrollierten Programmen zugeordnet werden können. Diese Prozessoren werden den Programmen unterteilt, darauf basierend wird die Anzahl der Prozesse pro Programm neu festgelegt. Bei der neu entstandenen Konfiguration ist die Gesamtanzahl der Prozesse gleich der Anzahl der Prozessoren im System. Für die vier analysierten Programme wächst die Beschleunigung bis zu 16 Prozessen für die im System kontrollierten und nicht-kontrollierten Programme, ohne daß der Mehraufwand der eingesetzten Technik signifikant wird. Dieses Verhalten war schon zu erwarten, da 16 Prozessoren zur Verfügung stehen. Ab 16 Prozessen ist die Beschleunigung für die kontrollierten Programme besser als für die nicht-kontrollierten, wobei dieser Unterschied größer wird, wenn die Gesamtanzahl der Prozesse wächst.

Diese Methode nimmt an, daß die Benutzung der maximalen Anzahl der Prozessoren zu den besten Ergebnissen führt und überprüft nicht, ob das der Realität entspricht. Außerdem, wenn die Gesamtanzahl der Prozesse im System zu hoch ist, bestimmt die zentrale Diensteinheit in einer sehr primitiven Art, welches Programm wieviel Prozesse beendet bzw. erzeugt, indem alle Programme eine möglichst gleiche Anzahl von Prozessen besitzen. Das kann aber zu schlechten Ergebnissen führen, wenn die Parallelitätsgrade der verschiedenen Programme sehr unterschiedlich sind.

Die als nächstes beschriebene Arbeit berücksichtigt die dynamische Zuordnung von Prozessoren zu Prozessen in Architekturen, die über verteilten Speicher verfügen.

Die Arbeit von **G. Edjlali, G. Agrawal, A. Sussman, J. Humphries und J. Saltz an der Universität Maryland** [14] hat eine dynamische bzw. adaptive parallele Umgebung als Ausgangspunkt. In solch einer Umgebung wird die Anzahl der zur Verfügung stehenden Prozessoren in der Zeit variiert. Das gesamte Konzept ist für ein Netz von Arbeitsplatzrechnern (*workstation*) realisiert, d.h. eine parallele Umgebung mit verteiltem Speicher, worauf in HPF (*High Performance Fortran* [32]) geschriebene Programme laufen. Der Parallelismus dieser Programme wird auf Schleifenebene betrachtet. Weil die parallele Umgebung adaptiv ist und verteilten Speicher hat, werden Aufgaben wie Datenumverteilung bei einer Änderung der Anzahl der zur Verfügung stehenden Prozessoren und Arbeitszuteilung während der Ausführungszeit notwendig. Der Mehraufwand für die Datenumverteilung kann hoch werden, falls solch eine Änderung sehr häufig auftritt. Genauso sollen die Laufzeitroutinen für die Arbeitzuteilung sehr effizient sein. Die Ergebnisse basieren auf einer nachrichtengekoppelten Umgebung, nämlich einem Netzwerk mit 12 Arbeitsplatzrechnern unter der Benutzung von PVM (*Parallel Virtual Machine* [15]) und auf einer IBM SP-2 mit 16 Prozessoren, wobei der Code von zwei Anwendungen dafür von Hand parallelisiert wurde. Die Ergebnisse zeigen, daß die Methode sehr schwer ihr Ziel erreicht, wenn der Parallelitätsgrad häufig eingestellt wird. Dies liegt daran, daß die Kommunikationszeit sehr hoch ist, besonders bei Arbeitsplatzrechnern, die durch Ethernet-Technologie verbunden sind.

Die Zielsetzung der o.a. Arbeit weicht wesentlich von der Zielsetzung der vorliegenden Arbeit ab. In der Arbeit von Edjlali et al. wird das Ziel verfolgt, daß ein paralleles Programm mit einer sich dynamisch ändernden Anzahl zur Verfügung stehender Prozessoren zurechtkommt, ohne zu berücksichtigen, ob diese Anzahl der Prozessoren für das Programm angebracht ist.

3.2 Verbesserung der Leistung paralleler Programme

Die zwei im folgenden beschriebenen Systeme befassen sich mit der Bestimmung des Parallelitätsgrades in Architekturen mit gemeinsamem Speicher und verfolgen das Ziel, parallele Programme effizient auszuführen.

Das System **REAPAR von S. Hänßgen an der Universität Karlsruhe** [25] präsentiert effiziente Parallelisierungsstrategien speziell für irreguläre rekursive Programme, die anstelle rekursiver Aufrufe parallel ablaufende Threads erzeugen. Die Zielarchitektur dabei sind Parallelrechner mit gemeinsamem Speicher. In ANSI C geschriebene Programme werden automatisch parallelisiert. Die Grundlage für die Auswahl der geeigneten Parallelisierungsstrategie des Programmes sind Daten, die in einem Programmlauf durch eine automatische Instrumentierung des Programmes gewonnen werden. Die Parallelisierungsstrategie bestimmt am Ort eines rekursiven Aufrufs, ob der Aufruf sequentiell oder mittels eines neuen Threads auszuführen ist. Sie soll genügend Threads erzeugen, um die Maschine auszulasten, aber nicht so viele oder so kleine, daß der Mehraufwand der Threadverwaltung die Effizienz beeinträchtigt. Beispiele für die Parallelisierungsstrategien sind *First N* (nur die ersten $N \times p$ Möglichkeiten, einen Thread zu erzeugen, werden wahrgenommen, wobei p der Anzahl der Prozessoren entspricht) und *Depth D* (erzeugt Threads genau in den oberen D Ebenen des Rekursionsbaums). Die Messungen wurden auf einer SUN SPARCstation 20 mit 4 HyperSPARC-Prozessoren (100 MHz) durchgeführt. Die damit erreichten Beschleunigungen liegen je nach Benchmark zwischen dem Faktor 2,8 und 4,0.

Sowohl die Zielarchitektur als auch die rekursive Art der Zielprogramme sind die wesentlichen Unterschiede zu der vorliegenden Arbeit. Zudem hat *REAPAR* die Parallelisierung rekursiver Programme als Ziel, während die vorliegende Arbeit von bereits parallelisierten Programmen ausgeht.

N. Reimer an der Universität Karlsruhe [44] hat sich im Rahmen seiner Diplomarbeit mit der dynamischen Einstellung des Parallelitätsgrades von Programmen auf Architekturen mit gemeinsamen Speichern befaßt. Er schlägt einen Ansatz zur Optimierung der Kosten (hier werden Kosten als die summierte Rechenzeiten über alle Prozessoren definiert) vor, bei dem ein paralleles Programm selbständig bei geringer Auslastung der Prozessoren diese teilweise freigibt bzw. bei hoher Last weitere Prozessoren belegt und die Arbeit jeweils umverteilt. Die Ausführungszeiten der parallelisierten Routinen werden für die Strategien der Einstellung verwendet. Die Strategien sind lokal oder global, je nachdem, ob alle möglichen Werte für den Parallelitätsgrad untersucht werden oder nur einige. Diese Strategien betrachten Kennzahlen wie z.B. Beschleunigung und Effizienz, die durch die parallelen Routinen erreicht werden. Eine Anwendung aus der Chemie (Moleküldynamik) wurde im Rahmen der Arbeit untersucht, wobei die Parallelisierung mittels PFA (*Power Fortran Accelerator*) stattgefunden hat. Die Hardware-Plattform bestand aus einer SGI Power Challenge mit 16 MIPS-R8000-Prozessoren, die die Möglichkeit anbietet, die Anzahl der verwendeten Prozessoren dynamisch anzupassen. Die Ergebnisse deuten darauf hin, daß die Strategie, die die Effizienz mit Vermeidung oszillierenden Verhaltens als Kriterium berücksichtigte, für das untersuchte Problem am besten war.[2] Die Kosten der Adaption mit dieser Strategie sind vergleichsweise gering gegenüber den Adaptionskosten mit globaler Suche. Keinerlei Vergleich der Beschleunigung bzw. Effizienz mit und ohne Einstellung des Parallelitätsgrades wurde angegeben.

Die Arbeit von N. Reimer am Institut für Programmstrukturen und Datenorganisation

[2] Bei einem oszillierenden Verhalten wird der Parallelitätsgrad wiederholend höher und dann niedriger eingestellt, so daß der Mehraufwand für die Einstellung des Parallelitätsgrades letztendlich sehr hoch ist.

an der Universität Karlsruhe war der Ausgangspunkt für die vorliegende Arbeit. Die Zielarchitektur, die Zielsprache, die Parameter, die die Einstellungsstrategien betreffen, und auch die Schnittstelle zwischen dem System und dem Benutzer bilden die Unterschiede zu der vorliegenden Arbeit. Um das System von N. Reimer anzuwenden, muß der Benutzer viele Änderungen in seinem ursprünglichen Programm vornehmen.

Allen B. Downey an der University of California, Berkeley [12] schlägt statistische Techniken zur Vorhersage von Wartezeiten (*queue times*) für Jobs in einem Mehrprozessorrechner vor und entwickelt Prozessorzuordnungs-Strategien, die diese Vorhersage benutzen. Anhand dieser Vorhersage entscheiden die Strategien, ob ein Job auf die Freigabe von zusätzlichen Prozessoren warten soll oder eher sofort auf bereits freistehenden Prozessoren laufen soll. Ziel der Arbeit ist die Minimierung der *turnaround time* eines Job, wobei die *turnaround time* durch die Summe der Wartezeit und der Ausführungszeit des Job gegeben ist. Hier werden adaptive bzw. *moldable* Jobs betrachtet, die zwar mit unterschiedlichen Prozessoranzahlen gestartet werden können, aber sobald die Ausführung anfängt, keine Änderung in der Anzahl der zugeordneten Prozessoren mehr erlauben. Um die Zeit bis zur Freigabe von *n'* Prozessoren ($Q(n')$, wobei $n' = n - n_{free}$; n ist die angefordete Prozessoranzahl, n_{free} ist die Anzahl der bereits freien Prozessoren) vorhersagen zu können, wird beispielsweise die *Median-Predictor*-Technik in der Arbeit vorgeschlagen. Durch diese Technik wird der Medianwert von $Q(n')$ so berechnet, daß alle Jobs betrachtet werden, die zu Ende ausgeführt wurden bzw. noch weiterlaufen. Anhand dieser Information wird die Wahrscheinlichkeit gebildet, daß n Prozessoren bis zu einem gegebenen Zeitpunkt t zur Verfügung stehen werden. Danach wird der Medianwert für die Wartezeit bestimmt. Die aus Simulationen hergeleiteten Ergebnisse deuten darauf hin, daß die verschiedenen, auf die Vorhersage der Wartezeit basierenden Strategien die Ausführungszeit der Jobs durchschnittlich um 12,8 bis 13,8 min reduzieren, wobei es nicht angegeben wurde, wieviel die durchschnittliche Ausführungszeit der Jobs beträgt.

Hier wird auf die Unterschiede der o.g. Arbeit zu der vorliegenden Arbeit näher eingegangen. Erstens schlägt die vorliegende Arbeit eine Lösung auf Programm- anstatt auf Systemebene vor. Zweitens wird die Anzahl der für die Ausführung eines Job eingesetzten Prozessoren bei der betrachteten Arbeit vor der Laufzeit festgelegt, während die vorliegende Arbeit dies anhand von Messungen dynamisch zur Laufzeit bestimmt. Drittens wird hier eine Verbesserung der Leistung des Programmes in dem Sinne erreicht, daß ein Kompromiß zwischen Wartezeit und Ausführungszeit getroffen wird. Es wird nicht überprüft, ob die Anzahl der Prozessoren, die letztendlich an der Ausführung eines Jobs teilnehmen, zu einer Minimierung der Ausführungszeit führt. Anders als bei der vorliegenden Arbeit ist man hier auf Zufall angewiesen.

3.3 Kombinierte Verbesserung der Auslastung und Leistung

Zu dieser Kategorie gehören Arbeiten, die einen Beitrag sowohl zur Verbesserung der Auslastung des Gesamtsystems als auch der Leistung paralleler Programme liefert. Die vorliegende Arbeit fällt selbst in diese Kategorie.

Die Arbeit von **M. Hall und M. Martonosi an den Universitäten Stanford bzw. Princeton** [24] hat eine effektive Zuordnung von Programmen, die durch den Stanford-SUIF-Übersetzer parallelisiert wurden, auf Parallelrechnern mit gemeinsamem Speicher zu Prozessoren als Ziel. Letztendlich sollen die parallelen Programme die Prozessoren ausnutzen

und eine gute Beschleunigung liefern können. Statt der eingeschränkten statischen Methode, die der SUIF-Übersetzer realisiert, schlägt die Arbeit ein adaptives Laufzeitsystem vor, das anhand dynamischer Information die Zuordnung von parallelen Schleifen zu Prozessoren in einem Mehrprozessorsystem mit gemeinsamem Speicher erledigt. Die Anzahl der Prozessoren, die jede parallele Schleife belegt, wird eingestellt. Realisiert wurde die Methode anhand von Threads, die über eine Menge von Programmen verteilt werden. D.h. die Programme, die aufgrund des Mangels an Parallelismus über zu viele Threads verfügen, müssen welche freigeben. Die freigegebenen Threads werden solchen Programmen zugeordnet, deren Leistung sich aufgrund von Messungen als gut bewiesen hat. Diese Messungen finden während der Laufzeit des Programmes statt und basieren auf der Beschleunigung vorheriger Ausführungen der Schleife. Die Ergebnisse wurden auf einer SGI-Power-Challenge-Maschine gesammelt, wobei alle Programme, die um die Threads konkurrieren, zusammen in ein einziges ausführbares Programm gebunden werden müssen. Die Ergebnisse deuten auf eine Verbesserung der Leistung der Programme bis auf 33% hin, wobei nur 5 der 29 Läufe mit der Thread-Verwaltung schlechtere Ergebnisse als eine *standalone*-Ausführung der parallelen Programme geliefert haben.

Nun werden noch die Abweichungen der Arbeit von M. Hall und M. Martonosi zu der vorliegenden Arbeit zusammengefaßt. Erstens sind die Zielarchitekturen unterschiedlich. Zweitens basieren die Strategien zur Einstellung des Parallelitätsgrades bei der vorliegenden Arbeit nicht nur auf der Beschleunigung in der parallelen Schleife, sondern auch auf der Effizienz, der Kosten-Nutzen-Relation und der Ausführungszeit. Drittens wird bei der Arbeit von M. Hall und M. Martonosi der Parallelismus im Programm mittels Threads (unterstützt die pseudo-parallele Ausführung mehrerer Prozesse) ausgenutzt, während bei der vorliegenden Arbeit die echtparallele Ausführung von Prozessen unterstützt ist, bei der ein Prozeß pro Prozessor läuft. Viertens sehen die Autorinnen keine Alternative für Programme vor, deren Einstellung des Parallelitätsgrades wegen Kommunikation nur mit sehr hohen Kosten möglich ist. Durch die vorliegende Arbeit wird der Parallelitätsgrad solcher Programme nach dem *inter-Lauf*-Verfahren eingestellt, d.h. über mehrere Läufe des Programmes. Interessant zu merken ist, daß obwohl die Autorinnen eine effektive Prozessorzuordnung als Arbeitsziel setzen, haben sie nur die Entwicklung der Beschleunigung von parallelen Programmen als Kriterium für die Einstellung des Parallelitätsgrades.

3.4 Einordnung und Vergleich der Arbeiten

Die folgende Tabelle gibt einen Überblick über die verwandten Arbeiten, für die eine Implementierung vorhanden ist. Zuerst werden die in Tabelle 3.1 benutzten Abkürzungen präsentiert.

Anzumerken ist, daß es keine Arbeit bis jetzt gab, die den Parallelitätsgrad dynamisch zur Laufzeit auf einem Mehrprozessorsystem mit verteiltem Speicher einstellt, und dabei sowohl die Verbesserung der Auslastung als auch der Leistung paralleler Programme unterstützt. Außerdem ist die Schnittstelle zwischen dem System und dem Benutzer als eine Bibliothek realisiert, wobei die Änderungen, die im Quellcode vorgenommen werden müssen, gering bleiben (der Mehraufwand in Code-Zeilen für die vier analysierten Anwendungen beträgt 2,3 bis 3,3% vom gesamten Programm).

Datenumverteilung:
+ vorhanden
− nicht vorhanden

Änderungsbedarf des Quellcodes:
0 keine
+ niedrig
++ hoch

Art der Parallelität:
P Prozesse
T Threads

Zielarchitektur:
G gemeinsamer Speicher
V verteilter Speicher

Ebene der Einstellung:
Sys Systemebene
Pro Programmebene

Beschaffung der Information für die Einstellung:
S statisch vorab
D dynamisch

Implementierung:
B Bibliothek
L Laufzeitsystem
Q Quellcode-Transformation

Tabelle 3.1: Einordnung der verwandten Arbeiten

Kriterium / Arbeit	Änderungsbedarf	Art der Parallelität	Zielarchitektur	Ebene der Einstellung	Beschaffung der Information	Implementierung	Datenumverteilung	Zielsprache
Vorliegende Arbeit	+	P	V	Pro	D	B,Q	+	C++
Reimer	++	P	G	Pro	D	B,Q	−	Fortran
M. Hall und M. Martonosi	0	T	G	Pro	D	L	−	C
M.S. Squillante et al.	0	T	V	Sys	S	L	−	∘[a]
Minos	0	T	G	Sys	S	L	−	C++
LLPC	+	T	G	Sys	S	Q,L	−	Fortran
A. Tucker und A. Gupta	0	T	G	Sys	D	L	−	∘[b]
G. Edjlali et al.	+	P	V	Sys	D	B,Q	+	HPF-ähnlich
REAPAR	+	T	G	Pro	D	B,Q	−	C
A. B. Downey	0	P	G	Sys	S	L	−	∘[a]

[a] Es handelt sich hier um eine Methode, die in das Betriebssystem eingebettet werden soll, so daß eine Zuordnung zu einer Zielsprache keine Rolle spielt. Die Zielsprache der Programme, die zu der Testmenge bei der präsentierten Ergebnisse gehören, wurde nicht angegeben.

[b] Die Zielsprache ist hier beliebig, vorausgesetzt, es steht eine *thread*-Bibliothek zur Verfügung, worin Änderungen zur Unterstützung der Methode vorgenommen werden müssen. Es wurde nicht angegeben, welche Programmiersprache für die Codierung der präsentierten Testmenge benutzt wurde.

Kapitel 4

Theoretische Modellbildung

In diesem Kapitel wird das im Rahmen dieser Arbeit entwickelte mathematische Modell zur Parallelitätsgradbestimmung präsentiert. Hier wird die Methode zur Ermittlung des optimalen Parallelitätsgrades vorgestellt und eine Klassifizierung paralleler Programme, auf die die Methode angewendet werden kann, vorgeschlagen. Danach folgt eine Analyse der Programmklassen in Hinblick auf die Ausführungszeit, die Effizienz und das Nutzen-Kosten-Verhältnis. Anschließend werden Erkenntnisse über das Amdal'sche Gesetz beim Nutzen-Kosten-Verhältnis vorgestellt.

Dieses Kapitel bildet zusammen mit Kapitel 5 das Lösungskonzept zu dem in Kapitel 1 eingeführten Problem der Einstellung des Parallelitätsgrades von Programmen.

4.1 Methode

Der optimale Parallelitätsgrad wird hier anhand der Ausführungszeit, der Effizienz oder der Nutzen-Kosten-Relation bestimmt. Gesucht wird der Parallelitätsgrad, für den die *Ausführungszeit* ihr Minimum, die *Effizienz* eine vorgegebene Unterschranke oder die *Nutzen-Kosten-Relation* ihr Maximum erreicht. In diesem Kapitel bilden die o.g. Ziele den Ausgangspunkt für die Modellbildung.

Die entwickelte Methode zur Parallelitätsgradbestimmung setzt voraus, daß die Funktionen für den Berechnungs- und Kommunikationsaufwand des Programmes, für das der optimale Parallelitätsgrad gefunden werden soll, bekannt sind. Anhand dieser Funktionen kann T_p, d.h. die Ausführungszeit eines Programmes mit p Prozessoren, durch die folgende Formel definiert werden:

$$\boxed{T_p = Berechnungsaufwand + Kommunikationsaufwand} \quad (4.1)$$

In der Regel treten auf der rechten Seite der Formel die Operanden T_1 (Ausführungszeit mit einem Prozessor), p (Anzahl der Prozessoren) und die Kommunikationszeit (siehe Abschnitt 2.4) auf. Die Kommunikationszeit ist wiederum möglicherweise Funktion von p.

Im Gegensatz zu der formalen Definition der Ausführungszeit (siehe Abschnitt 2.4) ist die Bereitschaftszeit in Formel 4.1 nicht enthalten. Dies beruht auf den folgenden Erkenntnissen: Erstens hat jeder Prozeß entweder etwas zu berechnen oder er kann terminiert werden, da die Benutzerprogramme lastbalanciert sind. Zweitens ist die für die Bereitstellung entfernter Daten notwendige Berechnung und Kommunikation im Kommunikationsaufwand durch die Latenzzeit (siehe Abschnitt 7.2.3) mit berücksichtigt.

Die Funktionen, die den Berechnungs- bzw. Kommunikationsaufwand modellieren, sollen möglichst genau sein, damit der Fehler bei der Berechnung des optimalen Parallelitätsgrades gering bleibt.[1]

Im folgenden werden die Verläufe der drei Kennzahlen, d.h. der Ausführungszeit, Effizienz und Nutzen-Kosten-Relation, als Funktion des Parallelitätsgrades erläutert. Außerdem betrachten wir die Bestimmung des optimalen Parallelitätsgrades auf Basis der Kennzahlen.

Die *Ausführungszeit* eines Programmes mit p Prozessoren wird auf die Formel 4.1 basierend durch die folgende Formel definiert:

$$T_p = \frac{T_1}{p} + T_{comm} \tag{4.2}$$

wobei T_1 die Ausführungszeit des Programmes mit einem Prozessor und T_{comm} die Kommunikationszeit wiedergibt.

Die Kurve der Ausführungszeit eines parallelen Programmes verläuft wie in Abbildung 4.1 dargestellt. Die Ausführungszeit sinkt bei zunehmender Prozessoranzahl bis zu einem Sättigungspunkt (Punkt K in Abbildung 4.1), ab dem sie wieder anfängt zu steigen. Das mathematische Modell soll genau den „Punkt K“, d.h. das Minimum dieser Kurve, bestimmen.

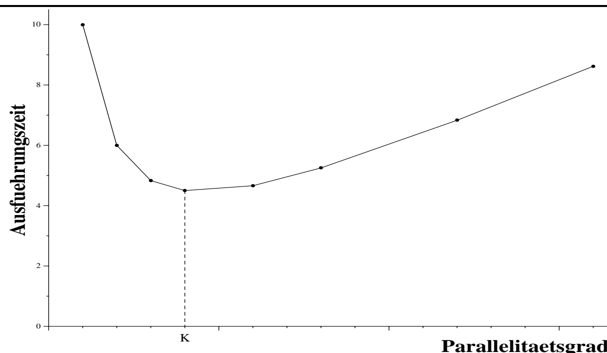


Abbildung 4.1: Kurve der Ausführungszeit eines parallelen Programmes

Das Minimum der Ausführungszeit wird gefunden, indem die Funktion, die die Ausführungszeit definiert, nach der Prozessoranzahl bzw. dem Parallelitätsgrad n abgeleitet und die Nullstelle dieser Ableitung ermittelt wird. Außerdem muß die zweite Ableitung der Funktion größer als Null sein, um zu gewährleisten, daß es sich um einen Tiefpunkt anstatt eines Hochpunktes handelt. Für einen Tiefpunkt gilt also:

$$T_p'(n) = 0 \quad und \quad T_p''(n) > 0 \tag{4.3}$$

Die *Effizienz* eines parallelen Programmes, das mit p Prozessoren läuft, ist wiederum durch die folgende Formel gegeben:

[1] Eine Studie (siehe Kapitel 6) hat gezeigt, daß das PRAM-Modell einen zu großen Fehler bei der Berechnung des optimalen Parallelitätsgrades zur Folge hat. Die von Ian Foster [21] vorgeschlagene Methode (siehe Abschnitt 2.5.4) hat sich als gut erwiesen.

$$E_p = \frac{T_1}{p \times T_p} \tag{4.4}$$

wobei T_1 die Ausführungszeit des Programmes mit einem Prozessor und T_p dieselbe mit p Prozessoren repräsentiert. Die Kurve der Effizienz eines parallelen Programmes hat den in Abbildung 4.2 gezeigten Verlauf.

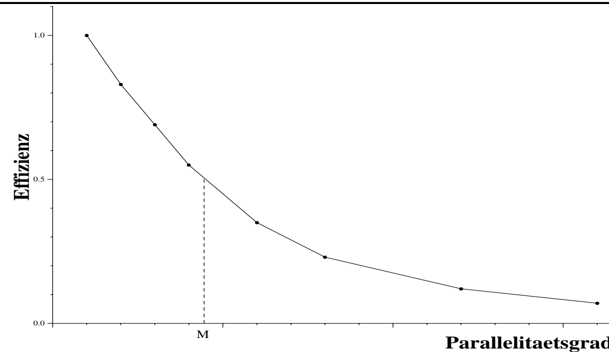


Abbildung 4.2: Kurve der Effizienz eines parallelen Programmes

Die Effizienz sinkt immer weiter bei zunehmendem Parallelitätsgrad. Durch das mathematische Modell wird der höchste Parallelitätsgrad bestimmt, für den die Effizienz einen vorgegebenen Wert erreicht. In dem in Abbildung 4.2 dargestellten Beispiel ist dieser vorgegebene Wert gleich 50% und somit der ermittelte Parallelitätsgrad gleich M.

Für den gesuchten optimalen Parallelitätsgrad, für den die Effizienz sich nicht unterhalb einer vorgegebenen Schranke befindet, gilt:

$$\frac{T_1}{p \times T_p} \geq Effizienzschranke \quad \Longrightarrow \quad p \leq \frac{T1}{Effizienzschranke \times T_p} \tag{4.5}$$

Die *Nutzen-Kosten-Relation* wird durch die folgende Formel definiert (wie in Abschnitt 2.4 detailliert erklärt):

$$R_{BC} = \frac{1}{R_{CB}} = \frac{T_1}{T_p^2 \times p} \tag{4.6}$$

wobei p die Anzahl der für die Ausführung des Programmes benutzten Prozessoren, T_1 die Ausführungszeit des Programmes mit einem Prozessor und T_p die Ausführungszeit mit p Prozessoren darstellt.

Die Kurve der Nutzen-Kosten-Relation eines parallelen Programmes verfolgt den in Abbildung 4.3 dargestellten Verlauf. Die Nutzen-Kosten-Relation steigt bei zunehmender Prozessoranzahl bis zu einem Sättigungspunkt (Punkt Q in Abbildung 4.3), ab dem sie wieder anfängt zu fallen. Genau der „Punkt Q“, d.h. das Maximum, wird hier durch das mathematische Modell bestimmt.

Analog zu dem Minimum der Ausführungszeit gilt für das Maximum der Nutzen-Kosten-Relation:

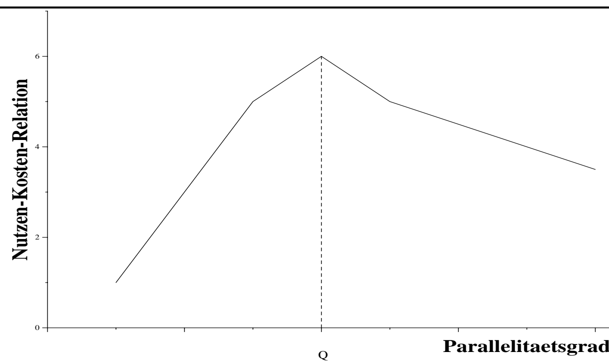


Abbildung 4.3: Kurve der Nutzen-Kosten-Relation eines parallelen Programmes

$$R'_{BC}(p) = 0 \quad und \quad R''_{BC}(p) < 0 \tag{4.7}$$

Unter der Benutzung des mathematischen Modells wird also der optimale Parallelitätsgrad je nach Ziel durch den Einsatz der Formel 4.2 und 4.3, 4.4 und 4.5 bzw. 4.6 und 4.7 ermittelt. Falls die Funktionen, die die Ausführungszeit bzw. die Nutzen-Kosten-Relation definieren, mehrere Minima bzw. Maxima haben, müssen diese noch miteinander verglichen werden, um das absolute Minimum bzw. Maximum zu bestimmen.

4.2 Klassifizierung von Programmen

Hier wird eine Klassifizierung von Programmen vorgeschlagen. Die Programme werden nach ihrem Kommunikationsverhalten gruppiert, wobei sich fünf Klassen herauskristallisiert haben:

- *Klasse K0* - Programme dieser Klasse führen keine Kommunikationsoperationen aus.
- *Klasse K1* - Programme dieser Klasse haben eine konstante Kommunikationszeit.
- *Klasse K2* - Kommunikationszeiten wachsen linear mit der Problemgröße bzw. mit der Anzahl von Prozessoren.
- *Klasse K3* - Kommunikationszeiten wachsen quadratisch mit der Problemgröße bzw. mit der Anzahl von Prozessoren.
- *Klasse K4* - Kommunikationszeiten wachsen logarithmisch mit der Problemgröße bzw. mit der Anzahl der Prozessoren.

4.3 Analyse von Programmklassen

Eine detaillierte Analyse des Verhaltens der Ausführungszeit, der Effizienz und der Nutzen-Kosten-Relation wird im folgenden anhand der o.g. Klassifizierung von Programmen vorgestellt.

4.3.1 Analyse in Hinblick auf die Ausführungszeit

Hier wird über die Entwicklung der Ausführungszeit anhand der unterschiedlichen Programmklassen diskutiert.

Klasse K0 – Wenn die Kommunikationszeit Null ($T_p = T_1/p$) ist, reduziert sich die Ausführungszeit immer weiter bei zunehmendem Parallelitätsgrad, so daß sich die minimale Ausführungszeit beim maximalen Parallelitätsgrad befindet. Dies wird in Abbildung 4.4 dargestellt.

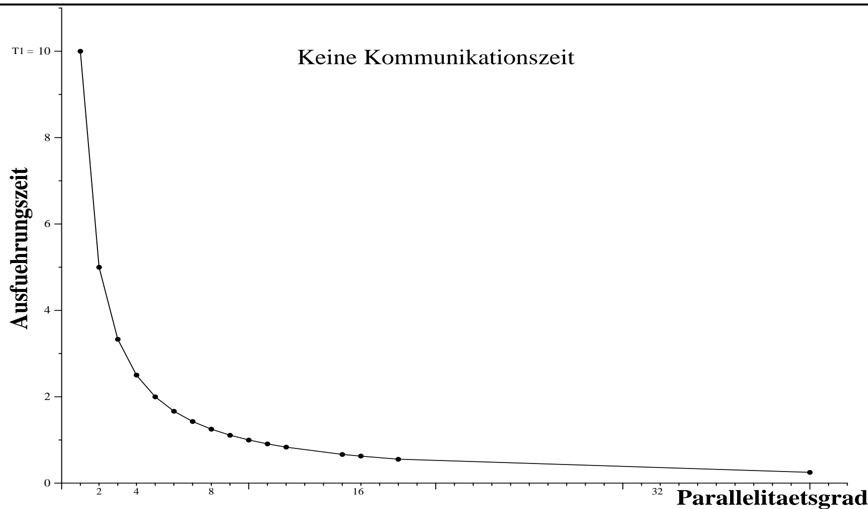


Abbildung 4.4: Ausführungszeit, wenn Kommunikationszeit = 0

Klasse K1 – Bei einem konstanten Kommunikationsaufwand ($T_p = T_1/p + c$) wird die Ausführungszeit eines parallelen Programmes zwar immer niedriger (siehe Abbildung 4.5), aber nicht optimal wie bei Programmen der Klasse K0. Hier hat die Funktion der Ausführungszeit mit p Prozessoren die Konstante c als Limes, während er bei Programmen der Klasse K0 gleich Null ist.

Klasse K2 – Falls der Kommunikationsaufwand linear zu der Prozessoranzahl wächst ($T_p = T_1/p + c \times p$, wobei c einer anwendungsabhängigen Konstante entspricht), fällt die Ausführungszeit bis zu einem gewissen Punkt und fängt dann wieder an zu steigen. Dieser Punkt entspricht dem Minimum der Ausführungszeit und ist bei dem Parallelitätsgrad $p = \sqrt{T1 \times c}/c$ gegeben. Abbildung 4.6 zeigt die Entwicklung der Ausführungszeit für diesen Fall, wobei $c = 0,10$ und $p = 10$.

Klasse K3 – Die Ausführungszeit eines Programmes wird durch die Formel $T_p = T_1/p + c \times p^2$ gegeben, wenn der Kommunikationsaufwand quadratisch wächst. Das Minimum dieser Funktion ist bei einem Parallelitätsgrad $p = \sqrt[3]{4}/2 \times \sqrt[3]{T1/c}$ zu finden. Hier ist das Minimum der Ausführungszeit tendenziell bei geringeren Werten der Prozessoranzahl (für $p = 9,6$ bei $c = 0,0055$ in Beispiel der Abbildung 4.7) als bei der Klasse K2 zu erkennen. Wie Abbildung 4.7 darstellt, steigt die Kurve nach dem Minimum sehr steil.

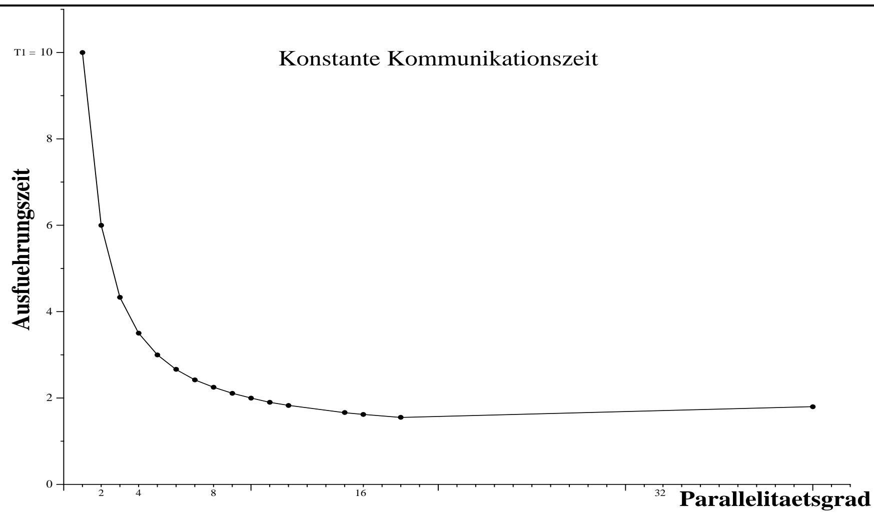


Abbildung 4.5: Ausführungszeit, wenn Kommunikationszeit konstant

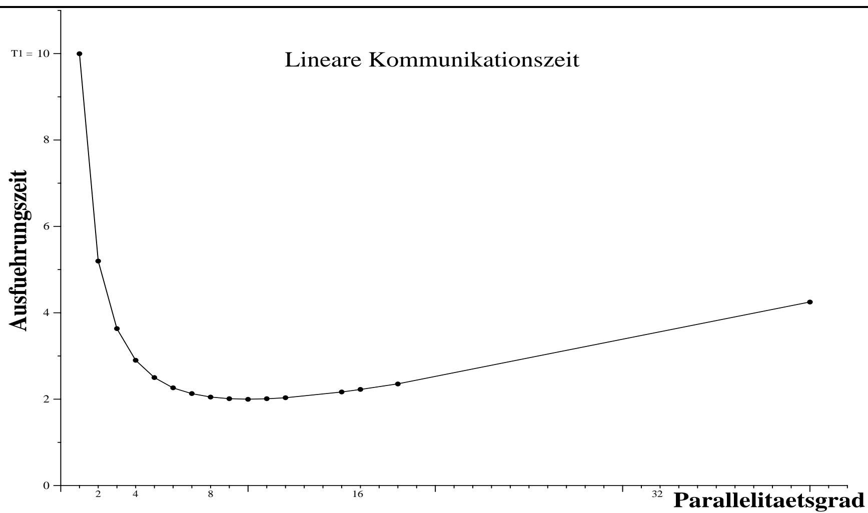


Abbildung 4.6: Ausführungszeit, wenn Kommunikationszeit linear

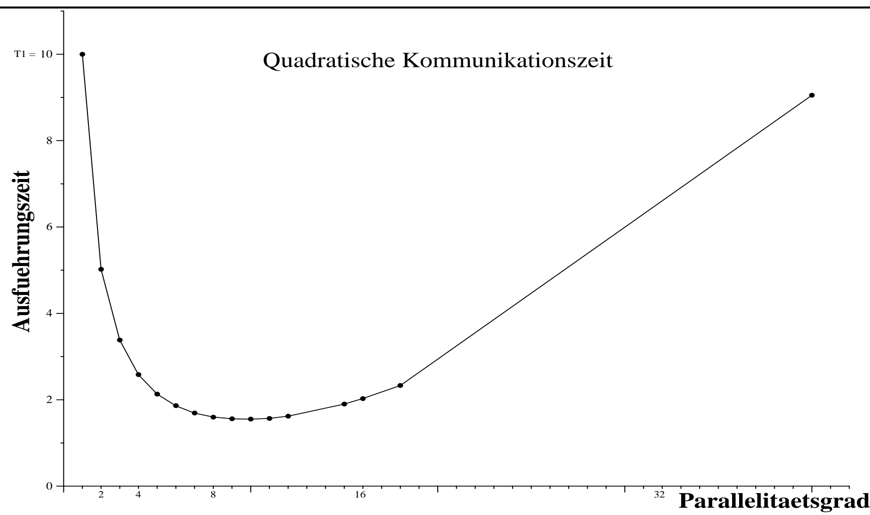


Abbildung 4.7: Ausführungszeit, wenn Kommunikationszeit quadratisch

Klasse K4 – Bei einer logarithmischen Kommunikationszeit wird die Ausführungszeit mit p Prozessoren durch die Formel $T_p = T_1/p + c \times log(p)$ gegeben (siehe Abbildung 4.8). Die anwendungsabhängige Konstante c spielt eine entscheidende Rolle bei der Bestimmung des Minimums der Funktion: Das Minimum befindet sich bei $p = T_1/c$. Bei dem Beispiel in Abbildung 4.8 wird das Minimum bei $p = 100$ gefunden, wobei $c = 0,1$.

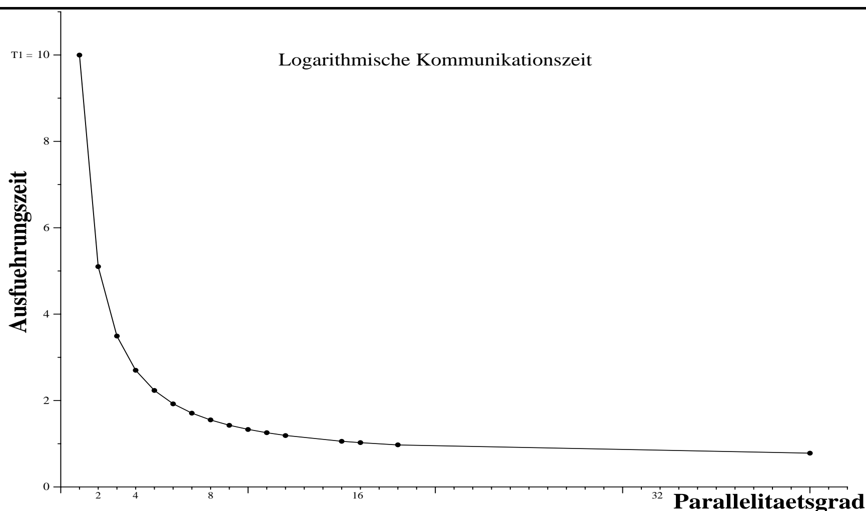


Abbildung 4.8: Ausführungszeit, wenn Kommunikationszeit logarithmisch

4.3.2 Analyse in Hinblick auf die Effizienz

Hier wird die Entwicklung der Effizienz anhand der unterschiedlichen Programmklassen vorgestellt. Die Diskussion der Effizienz aller Programmklassen basiert auf Abbildung 4.9, die den Verlauf der Effizienz für die unterschiedlichen Programmklassen bzw. Kommunikationsverhalten darstellt.

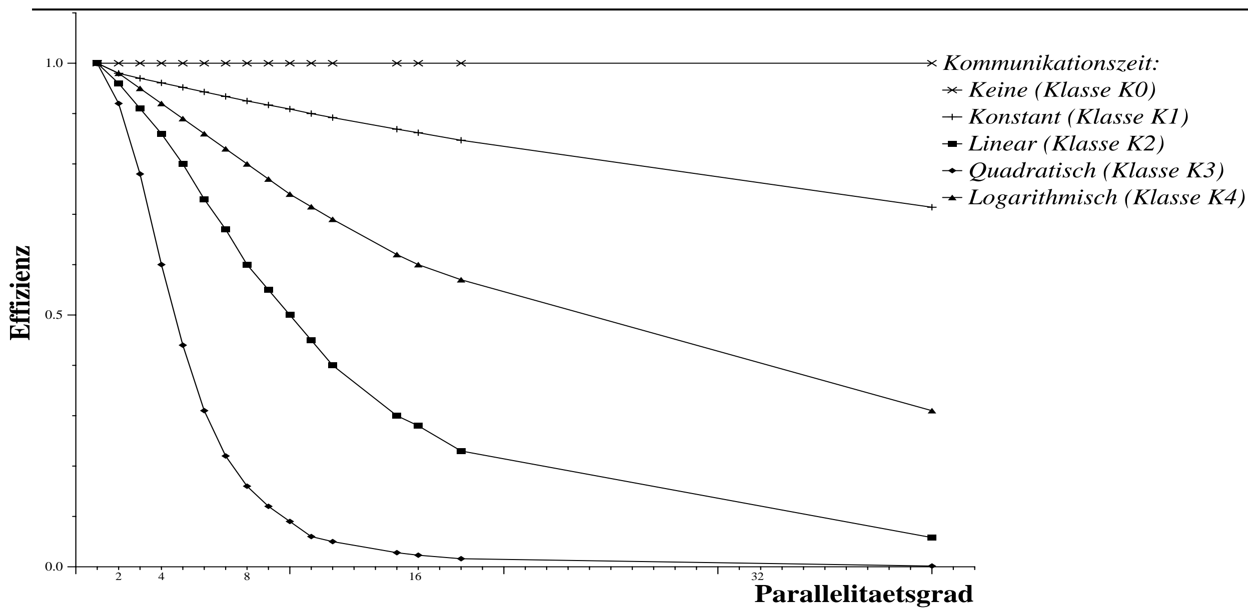


Abbildung 4.9: Effizienz-Verläufe der Programmklassen

Klasse K0 – Die Effizienz eines parallelen Programmes bleibt weiterhin optimal, unabhängig davon, wieviele Prozessoren an der Ausführung des Programmes beteiligt sind, wenn die Kommunikationszeit gleich Null ist. Da $T_p = T_1/p$, gilt basierend auf der Formel 4.5 für jede Prozessoranzahl: $1 \geq Effizienzschranke$.

Klasse K1 – Wenn der Kommunikationsaufwand konstant ist, wird die Ausführungszeit durch die Formel $T_p = T_1/p + c$ gegeben und somit gilt für den gesuchten optimalen Parallelitätsgrad $p \leq (T_1 \times (1 - E))/(E \times c)$, wobei c der konstanten Kommunikationszeit und E der Effizienzschranke entspricht. Abbildung 4.9 zeigt, daß die Effizienz in diesem Fall immer weiter so steil fällt, wie durch die Konstante c (im Beispiel gleich 0,1) bestimmt. Hier ist $p \leq 25$, wenn die Effizienzschranke gleich 80% ist.

Klasse K2 – Im Fall einer linearen Kommunikationszeit ($T_p = T_1/p + c \times p$) fällt die Effizienz bei gleicher Konstante c noch steiler als bei einem konstanten Kommunikationsaufwand. Das passiert in dem in Abbildung 4.9 dargestellten Beispiel ab $p = 10$, wobei $c = 0,1$. Gesucht wird der maximale Parallelitätsgrad, für den $p \leq \sqrt{(T_1 \times (1 - E))/(E \times c)}$ gilt, wobei E der Effizienzschranke entspricht. Für das Beispiel in Abbildung 4.9 ist $p \leq 20$, wenn die Effizienzschranke 20% beträgt.

Klasse K3 – Bei einer quadratischen Kommunikationszeit sinkt die Effizienz noch stärker als bei der Klasse K2. Der gesuchte Parallelitätsgrad ist die maximale, für den $p \leq \sqrt[3]{(T_1 \times (1 - E))/(E \times c)}$ gültig ist, wobei E die Effizienzschranke repräsentiert. Für das entsprechende Beispiel in Abbildung 4.9 ist $p \leq 7$, wenn die Effizienzschranke gleich 20% und $c = 0,1$ ist.

Klasse K4 – Die Effizienz fällt nicht so steil wie bei der Klasse K2 bzw. K3 bei gleicher Konstante c ab, wenn die Kommunikationszeit ein logarithmisches Verhalten hat. Abbildung 4.9 zeigt die Entwicklung der Effizienz für ein Beispiel, bei dem $c = 0,1$. Für eine gegebene

Effizienzschranke gilt $(T_1/(T_1 + c \times p \times log(p))) \geq Effizienzschranke$. Diese Ungleichung kann numerisch gelöst werden, wobei $p \leq 8$ bei einer Effizienzschranke von 80% gilt.

4.3.3 Analyse in Hinblick auf das Nutzen-Kosten-Verhältnis

Das Verhalten der Nutzen-Kosten-Relation bei Grenzfällen ist in Abbildung 4.10 zu sehen. Bei dem *best-case*, d.h. keine Kommunikationszeit ist vorhanden ($T_p = T_1/p$, wobei T_p bzw. T_1 die Ausführungszeit mit p Prozessoren bzw. mit einem Prozessor repräsentieren) wächst die Nutzen-Kosten-Kurve immer weiter bei zunehmender Prozessoranzahl, so daß das Optimum durch den maximalen Parallelitätsgrad bestimmt wird, wie in Abbildung 4.10 durch den Bereich *steigende Tendenz* angedeutet. Bei dem *worst-case*, bei dem der Einsatz mehrerer Prozessoren zu keiner Verbesserung der Ausführungszeit im Vergleich zu dem Fall mit einem einzigen Prozessor führt ($T_p = T_1$), wird die Nutzen-Kosten-Relation immer geringer bei einer wachsenden Prozessoranzahl, so daß das Maximum bei Parallelitätsgrad gleich 1 liegt, wie in Abbildung 4.10 im Bereich *abfallende Tendenz* gezeigt.[2] Bei einer Ausführungszeit mit p Prozessoren, bei der $T_p = T_1/\sqrt{p}$, bleibt die Nutzen-Kosten-Relation bei einer steigenden Anzahl von Prozessoren konstant. Sie wird durch den Kehrwert der Ausführungszeit mit einem Prozessor gegeben (horizontale Linie in Abbildung 4.10). Basierend auf dieser Erkenntnis gehört auch dieser Fall zu der abfallenden Tendenz des optimalen Parallelitätsgrades.

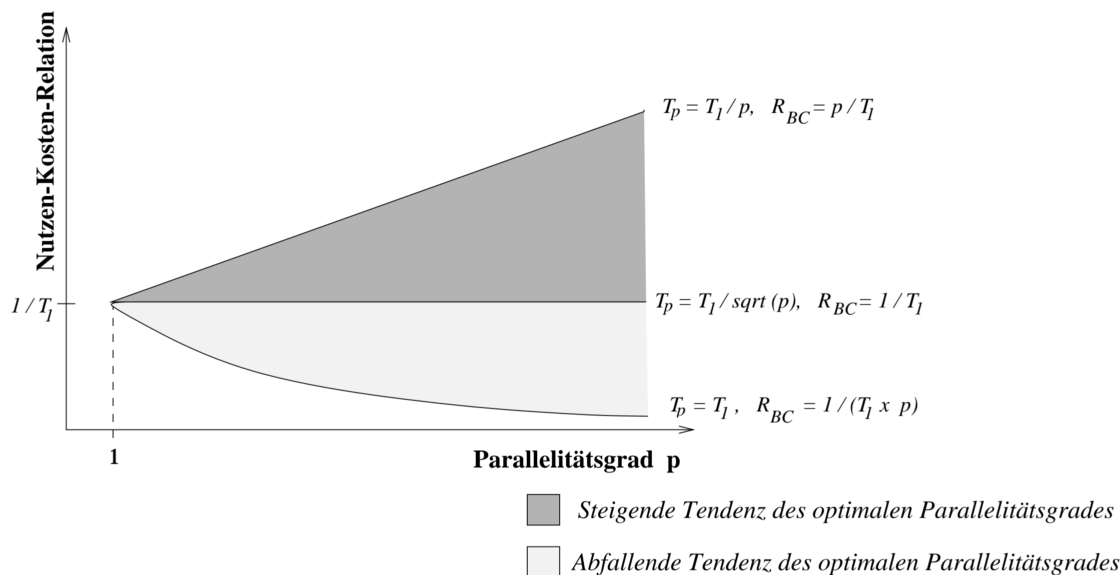


Abbildung 4.10: Tendenz des optimalen Parallelitätsgrades anhand der Nutzen-Kosten-Relation

Bei der Analyse der verschiedenen Programmklassen auf das Nutzen-Kosten-Verhältnis hin wird hier angenommen, daß es keinerlei redundanten Code bei der parallelen Ausführung eines Programmes gibt, so daß die Berechnungszeit immer durch den Quotienten T_1/p gege-

[2] Die Festlegung von $T_p = T_1$ als *worst-case* ist eine Vereinfachung, da $T_p > T_1$ auch möglich ist, aber nicht betrachtet wird. Das Verhalten der Nutzen-Kosten-Relation wäre aber auch im zweiten Fall vergleichbar, nur daß die Nutzen-Kosten noch steiler fallen würde. Das Optimum läge auch beim Parallelitätsgrad 1.

ben ist. Der Berechnungsaufwand eines jeden Prozessors bei einer Ausführung des parallelen Programmes mit p Prozessoren wird folgendermaßen definiert:

$$T_p = \frac{T1}{p} + T_{comm}$$

Im folgenden wird über die Entwicklung der Nutzen-Kosten-Verhältnis anhand der unterschiedlichen Programmklassen diskutiert.

Klasse K0 – Im Falle der optimalen Ausführungszeit (d.h. wenn die Kommunikationszeit gleich Null ist) wird die Ausführungszeit mit p Prozessoren, die durch die Formel $T_p = T_1/p$ gegeben ist, bei steigender Anzahl von Prozessoren stark verringert. Folge daraus ist, daß die Nutzen-Kosten-Relation, die in diesem Fall durch $R_{BC} = p/T_1$ dargestellt wird, immer weiter mit der Anzahl der Prozessor steigt, wie in Abbildung 4.11 zu sehen ist. Das Maximum der Nutzen-Kosten-Relation, d.h. der optimale Parallelitätsgrad, liegt in diesem Fall bei der maximalen Prozessoranzahl.

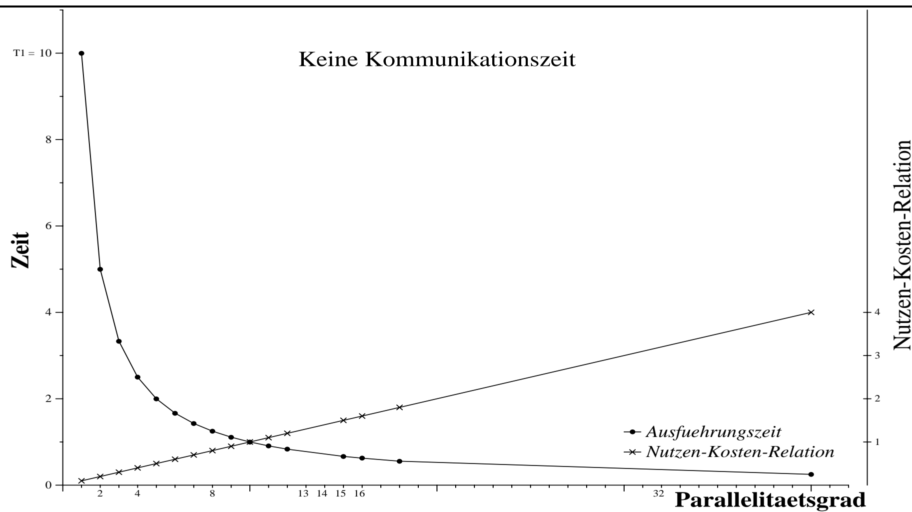


Abbildung 4.11: Nutzen-Kosten-Relation, wenn keine Kommunikationszeit

Klasse K1 – Die Ausführungszeit eines Programmes mit p Prozessoren und einem konstanten Kommunikationsaufwand c ($T_{comm} = c$) ist durch die Formel $T_p = (T_1/p) + c$ gegeben. Basierend auf der Formel (4.6) ergibt sich für die Nutzen-Kosten-Relation:

$$R_{BC} = \frac{T_1 \times p}{(T_1 + c \times p)^2}$$

Abbildung 4.12 zeigt die Ausführungszeit- bzw. die Nutzen-Kosten-Kurve für diesen Fall, wobei $c = 1$.

Hier wird die Ausführunszeit bei steigender Prozessoranzahl immer weiter reduziert. Jedoch lohnt es sich aufgrund der Nutzen-Kosten-Relation nicht, die Prozessoranzahl immer weiter zu steigern. Es gibt einen Punkt, nämlich für $p = T_1/c$ (bei einer Prozessoranzahl gleich 10 in dem in Abbildung 4.12 dargestellten Beispiel), ab den die Nutzen-Kosten-Relation wieder niedriger wird. Die Stelle dieses Maximums wird gesucht und ist von der Konstante c abhängig. Je größer c ist, desto früher wird das Maximum erreicht.

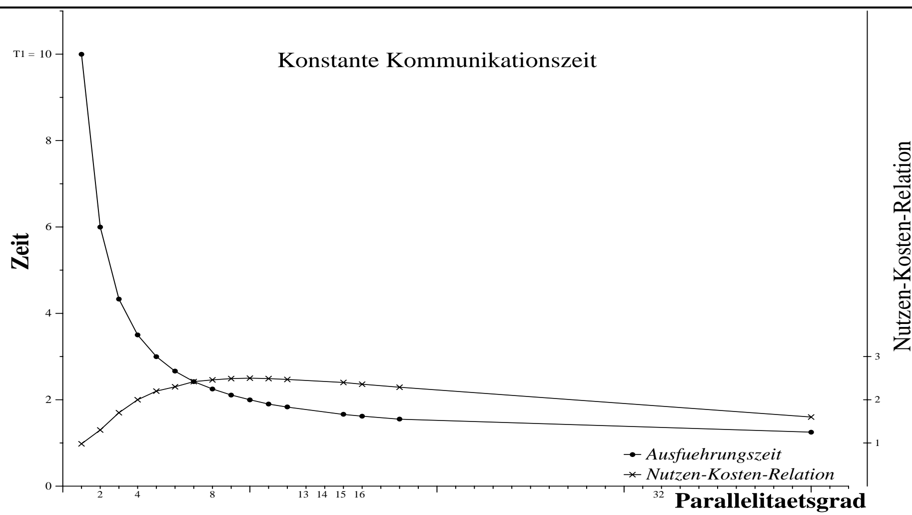


Abbildung 4.12: Nutzen-Kosten-Relation bei konstanter Kommunikationszeit

Klasse K2 – Im Fall eines linearen Kommunikationsaufwandes ($T_{comm} = c \times p$) ist die Ausführungszeit eines parallelen Programmes mit p Prozessoren durch die Formel $T_p = (T_1/p) + c \times p$ gegeben, wobei c einer anwendungsabhängigen Konstante entspricht. Die Nutzen-Kosten-Relation wird folgendermaßen definiert:

$$R_{BC} = \frac{T_1 \times p}{(T_1 + c \times p^2)^2}$$

Abbildung 4.13 zeigt die Entwicklung der Ausführungszeit bzw. der Nutzen-Kosten-Relation für diesen Fall, wobei $c = 0,10$.

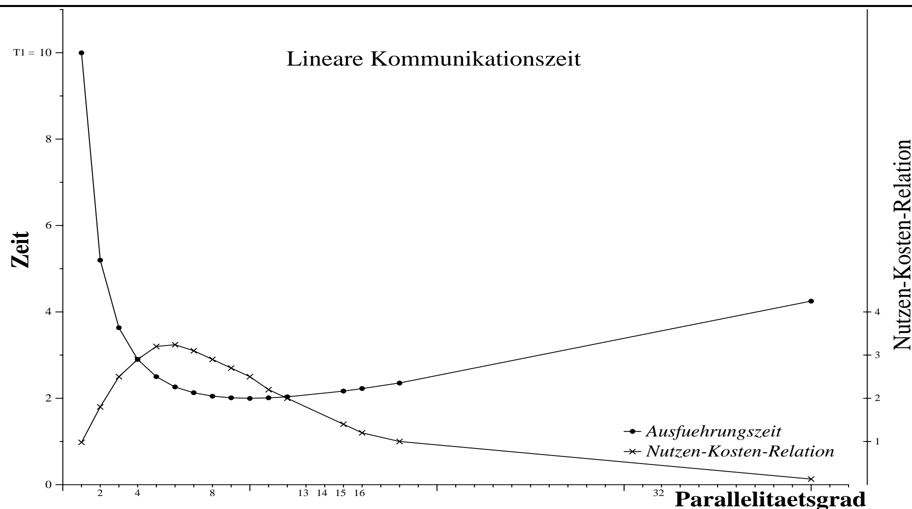


Abbildung 4.13: Nutzen-Kosten-Relation bei linearer Kommunikationszeit

Anzumerken ist, daß die Ausführungszeit bei zunehmender Prozessoranzahl bis zu einem gewissen Punkt fällt, und dann wieder anfängt so zu steigen, daß die Steilheit der Kurve durch die Konstante c definiert ist. Die Nutzen-Kosten-Kurve ist nicht mehr so flach wie bei

Programmen der Klasse K1; sie wächst dagegen steiler, trifft das Maximum früher, d.h. bei einer geringeren Prozessoranzahl, nämlich bei $p = (\sqrt{3 \times c \times T_1})/(3 \times c)$, und fällt wieder steiler als bei der Klasse *K1*.

Dieses Verhalten entspricht der Erwartung, weil die Kommunikationszeit hier tendenziell höher ist als bei der Klasse K1. Folge daraus ist, daß die Verbesserung der Ausführungszeit bei zunehmender Prozessoranzahl signifikanter sein müßte, damit die Nutzen-Kosten-Relation immer weiter dafür sprechen würde, daß mehr Prozessoren hinzugenommen werden sollten. Die Tendenz ist, daß das Maximum der Nutzen-Kosten-Relation, im Vergleich zu der Klasse K1, bei kleineren Werten der Prozessoranzahl liegt. Für das Beispiel in Abbildung 4.13 passiert das bei $p = 5,77$.

Klasse K3 – Die Ausführungszeit eines parallelen Programmes mit p Prozessoren wird duch die Formel $T_p = (T_1/p) + c \times p^2$ für den Fall einer quadratischen Kommunikationszeit ($T_{comm} = c \times p^2$) gegeben, wobei c einer anwendungsabhängigen Konstante entspricht. Die Nutzen-Kosten-Relation ergibt sich in diesem Fall:

$$R_{BC} = \frac{T_1 \times p}{(T_1 + c \times p^3)^2}$$

Abbildung 4.14 zeigt das Verhalten der Ausführungszeit bzw. der Nutzen-Kosten-Relation in diesem Fall, wobei $c = 0,0055$.

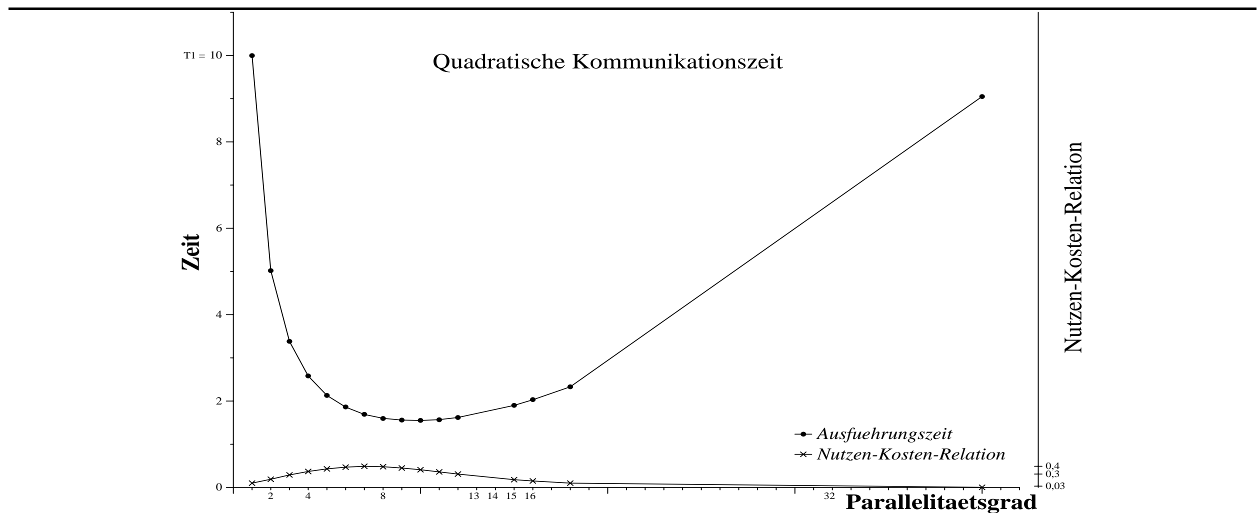


Abbildung 4.14: Nutzen-Kosten-Relation bei quadratischer Kommunikationszeit

Hier ist das Minimum der Ausführungszeit tendenziell bei noch geringeren Werten der Prozessoranzahl als bei der Klasse K2 zu erkennen. Außerdem steigt die Kurve noch steiler als bei der Klasse K2. Folge daraus ist, daß das Maximum der Nutzen-Kosten-Relation auch bei einem geringeren Parallelitätsgrad als bei der Klasse K2, nämlich bei $p = \sqrt[3]{25}/5 \times \sqrt[3]{T1/c}$, zu finden ist, wenn c derselbe ist.

Klasse K4 – Bei einem logarithmischen Verhalten der Kommunikationszeit ($T_{comm} = c \times log(p)$) ist die Ausführungszeit eines parallelen Programmes mit p Prozessoren durch

$T_p = (T_1/p) + c \times log(p)$ gegeben, wobei c einer Konstante entspricht. Für die Nutzen-Kosten-Relation ergibt sich in diesem Fall:

$$R_{BC} = \frac{T_1 \times p}{(T_1 + c \times p \times log(p))^2}$$

Abbildung 4.15 zeigt die Entwicklung der Ausführungszeit bzw. der Nutzen-Kosten-Relation beim logarithmischen Kommunikationsaufwand.

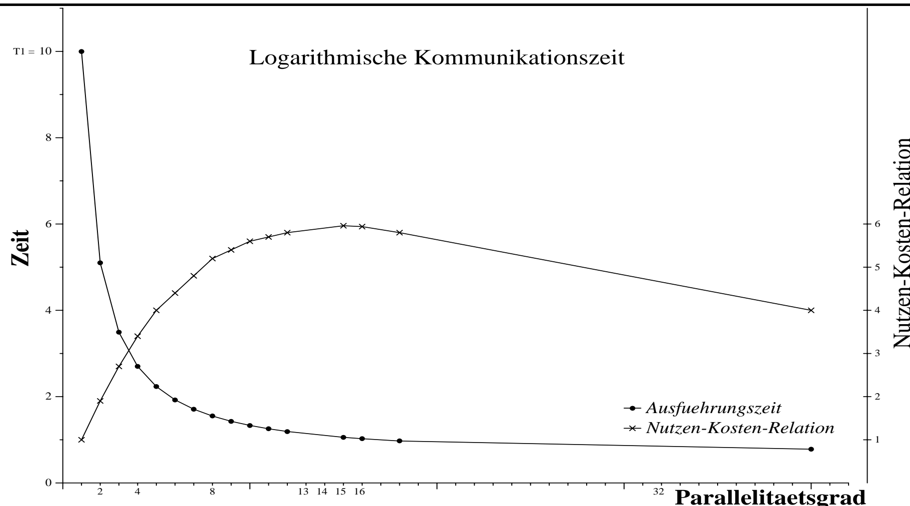


Abbildung 4.15: Nutzen-Kosten-Relation bei logarithmischer Kommunikationszeit

Hier wird die Ausführunszeit bei steigender Prozessoranzahl immer weiter reduziert, aber jedoch nicht so, daß es sich aufgrund der Nutzen-Kosten-Relation lohnen würde, die Prozessoranzahl immer weiter zu erhöhen. Es gibt hier einen Punkt (bei ca. 16 Prozessoren bei dem Beispiel in Abbildung 4.15), bei dem die Nutzen-Kosten-Relation wieder niedriger wird. Nach diesem Punkt führt eine Hinzunahme weiterer Prozessoren nicht mehr zu einer bedeutenden Verbesserung der Ausführungszeit, im Vergleich zu der Verschlechterung der Auslastung des Gesamtrechners.

4.4 Das Amdahl'sche Gesetz im Nutzen-Kosten-Verhältnis

Hier wird eine zu Alternative zu der im Abschnitt 4.1 beschriebenen Methode präsentiert. Anstatt Funktionen für die Berechnungs- und Kommunikationsaufwände, müssen hier α und β bekannt sein.

Wie im Abschnitt 2.5.1 ausführlich diskutiert wurde, besteht ein paralleler Algorithmus aus einem parallelen Teil α und einem sequentiellen Teil $\beta = 1 - \alpha$. Daher ist die Ausführungszeit des parallelen Programmes mit einem Prozessor durch die Formel $T_1 = \alpha + \beta$ und mit p Prozessoren durch $T_p = \alpha/p + \beta$ gegeben. Auf Basis der Formel $R_{BC} = T_1/(T_p^2 \times p)$ läßt sich die Nutzen-Kosten-Relation durch das Amdahl'sche Gesetz berechnen:

$$R_{BC} = \frac{T_1 \times p}{(\alpha + \beta \times p)^2}$$

Um den optimalen Parallelitätsgrad (p) zu berechnen, d.h. um das Maximum der Nutzen-Kosten-Relation zu bestimmen (siehe Abschnitt 4.1) wird R_{BC} nach p abgeleitet:

$$R'_{BC}(p) = \frac{-T_1(-\alpha + \beta \times p)}{(\alpha + \beta \times p)^3}$$

und die Nullstelle dieser Ableitung berechnet:

$$-T_1(-\alpha + \beta \times p) = 0$$

Für den optimalen Parallelitätsgrad erhalten wir:

$$p_{opt} = \frac{\alpha}{\beta} = \frac{\alpha}{1 - \alpha}$$

Die Nutzen-Kosten-Relation erreicht ihr Maximum bei einer Prozessoranzahl, die dem Verhältnis zwischen dem parallelen und dem sequentiellen Teil entspricht. Daraus folgt, daß wenn der sequentielle Teil des Programmes z.B. 10% beträgt, ist der optimale Parallelitätsgrad anhand der Nutzen-Kosten-Relation gleich 9. Eine interessante Beobachtung ist, daß wenn $\beta > \alpha$, d.h. der sequentielle Teil größer ist als der parallele, wird das maximale Nutzen-Kosten-Verhältnis immer bei einer Prozessoranzahl gleich 1 erreicht.

Weil das Amdahl'sche Gesetz die Kommunikationsoperationen zwischen Prozessoren nicht berücksichtigt, ist die o.g. Erkenntnis nur für Programme der Klasse K0 gültig, d.h. Programme, die nicht kommunizieren. Diese Erkenntnis kann aber für die verschiedenen Programmklassen verallgemeinert werden, indem in der Ausführungszeit mit p Prozessoren die Kommunikationszeit berücksichtigt wird. Dies ist durch die folgende Formel gegeben:

$$T_p = \frac{\alpha}{p} + \beta + T_{comm}$$

Die Kommunikationszeiten der verschiedenen Klassen (siehe Abschnitt 4.2) können eingesetzt werden, um den optimalen Parallelitätsgrad zu berechnen. Als Beispiel betrachten wir die Klasse K1. Die Kommunikationszeit T_{comm} ist konstant, d.h. $T_p = \alpha/p + \beta + c$. Das Maximum der Nutzen-Kosten-Relation ist in diesem Fall durch die folgende Formel gegeben:

$$p = \frac{\alpha}{1 - \alpha + c}$$

Hier wird der optimale Parallelitätsgrad nicht nur durch den sequentiellen Anteil des Programmes erniedrigt, sondern auch durch die Konstante, die die Kommunikationszeit festhält.

4.5 Diskussion der theoretischen Modellbildung

In der vorliegenden Arbeit diente das mathematische Modell, zusätzlich zu der Bestimmung des optimalen Parallelitätsgrades (siehe Abschnitt 7.2), auch als Hilfsmittel zur Verifizierung (siehe Abschnitt 7.4) des Modells zur automatischen Einstellung des Parallelitätsgrades (siehe Kapitel 5). Das mathematische Modell hat zusätzlich als Basis zu einer Studie des Verhaltens der Kurven der Ausführungszeit, der Effizienz und der Nutzen-Kosten-Relation gedient

und dadurch gezeigt, daß die automatische Einstellung des Parallelitätsgrades möglich ist bzw. daß die gesuchten Optima zu finden sind.

Wie im Abschnitt 4.1 bereits erwähnt, die Voraussetzung für den Einsatz des mathematischen Modells zur Ermittlung des optimalen Parallelitätsgrades ist die Verfügbarkeit des Berechnungs- und Kommunikationsaufwandes der zu betrachtenden parallelen Programme. Bei solchen parallelen Programmen, für die es keine genauen Angaben über den Berechnungs- bzw. Kommunikationsaufwand gemacht werden kann, kann das mathematische Modell anhand einer Approximierung der o.g. Aufwände für die Annäherung an den optimalen Parallelitätsgrad eingesetzt werden. Die ermittelte Annäherung wird als Initialwert für den Parallelitätsgrad bei der automatischen Einstellung benutzt[3]. Dadurch läßt sich der optimale Parallelitätsgrad u.U. schneller finden als wenn von dem vom Benutzer bzw. vom Betriebssystem angegebenen Initialwert für die Prozessoranzahl ausgegangen wird.

[3]Für das Programm LL3 (siehe Abschnitt 7.2.3.6) ist 128 der über das mathematische Modell ermittelte Initialwert für den Parallelitätsgrad, wobei der reale optimale Parallelitätsgrad 16 beträgt.

Kapitel 5

Automatische Bestimmung des Parallelitätsgrades

Hier wird das im Rahmen der vorliegenden Arbeit entwickelte Modell zur Bestimmung des optimalen Parallelitätsgrades eines Programmes präsentiert. Wie bereits in Abschnitt 1.2 erläutert, dient das Modell zum Beweisen der dort eingeführten Thesen. Es wurden zwei Methoden entwickelt, die den optimalen Parallelitätsgrad ermitteln. Die erste Methode beschäftigt sich mit der Bestimmung des Parallelitätsgrades über ein mathematisches Modell und wurde im Kapitel 4 erläutert. Die zweite Methode betrachtet die automatische Einstellung des Parallelitätsgrades mittels eines Suchverfahrens und ermöglicht die Einstellung des Parallelitätsgrades auch für solche Programme, für die der Berechnungs- bzw. Kommunikationsaufwand nicht gegeben werden kann. Sie wird im folgenden vorgestellt.

Zuerst werden die Randbedingungen und Aufgaben erläutert. Danach werden ein Szenario der Einstellung des Parallelitätsgrades, das eingesetzte Suchverfahren zur Parallelitätsgradbestimmung, die Strategien für die Einstellung des Parallelitätsgrades, die benötigten Laufzeitinformationen, der Datenumverteilungsmechanismus, die unterschiedlichen Arten der Einstellung des Parallelitätsgrades sowie eine Parallelitätsgrad-Datenbank vorgestellt.

5.1 Randbedingungen und Aufgaben

Die folgenden Randbedingungen sind bei der Arbeit zu berücksichtigen:

- Die Anpassung des Parallelitätsgrades soll auf Programmebene und nicht auf Systemebene geschehen.[1] Dadurch ergeben sich zwei Vorteile. Erstens sind die Portierbarkeit und die Kompatibilität der Programme gewährleistet, weil keine Änderungen am Betriebssystem vorgenommen werden müssen. Zweitens kann der Benutzer einen direkten Einfluß auf das Auswählen der Stellen haben, an denen der Parallelitätsgrad angepaßt werden soll. Das Parallelitätsprofil eines fiktiven Programmes, das über vier parallelisierbare Schleifen verfügt, ist in Abbildung 5.1 gezeigt. Sinnvoll ist, den Parallelitätsgrad an den Stellen a, c und d einzustellen. Nachdem die Einstellung des Parallelitätsgrades am Punkt a stattgefunden hat, würde ein Versuch, den Parallelitätsgrad an dem Punkt b anzupassen, nur zusätzliche Kosten verursachen.

[1] Das ist ein wesentlicher Unterscheidungspunkt zwischer der vorliegenden Arbeit und den Ansätzen, die sich mit der Prozessorzuteilung auf Parallelrechnern auseinandersetzen.

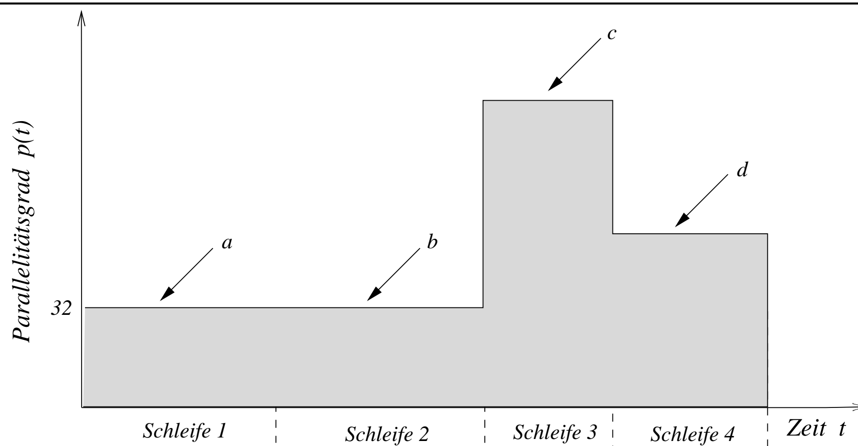


Abbildung 5.1: Parallelitätsprofil eines fiktiven Programmes

- Die Einstellung des Parallelitätsgrades findet auf Schleifenebene statt. Der Benutzer spezifiziert die Stellen, an denen die Einstellung geschehen soll. Der Parallelitätsgrad der *Schleife 1* in Abbildung 5.1 beträgt 32. Angenommen, der Benutzer hat das Programm mit 128 Prozessoren gestartet, so soll diese innerhalb der Schleife auf 32 reduziert werden. Das kann aber zu der Idee führen, daß die Problemgröße innerhalb der Schleife konstant bleiben muß, was nicht der Fall ist. Das Modell ist auch für solche Probleme anwendbar, deren Größe in der Zeit variiert. Die Einstellung wird von Zeit zu Zeit zurückgesetzt, nämlich jedesmal, wenn sich die Problemgröße ändert. Beim Zurücksetzen erhält die Prozessoranzahl ihren Initialwert, nämlich den vom Benutzer bzw. vom Betriebssystem gesetzten Wert.

- Die für die Einstellung des Parallelitätsgrades benötigten Informationen werden dynamisch zur Laufzeit des Programmes gewonnen. Es folgt daraus, daß keine Unterstützung zur Übersetzungszeit gebraucht wird. Vorteil daran ist, daß die Einstellung realitätsnah bleibt, denn Aspekte wie z.B. Granularität, Problemgröße und Kommunikationszeit werden oft erst während der Ausführungszeit bekannt. Eine statische Abschätzung solcher Größen sowie die Berechnung des Parallelitätsgrades auf deren Basis kann leicht zu ungeeigneten Werten für den Parallelitätsgrad führen. Ein weiterer Vorteil ist, daß keine Änderungen am Übersetzer vorgenommen werden müssen.

- Der Parallelitätsgrad von Programmen soll so automatisch wie möglich, d.h. mit möglichst wenig Einfluß des Benutzers, eingestellt werden.

- Der Benutzer des Systems soll den Quellcode seiner Programme so wenig wie möglich ändern brauchen, um den Parallelitätsgrad automatisch anpassen zu können.

- Das Programm darf innerhalb der parallelisierbaren Schleifen, für die eine Einstellung des Parallelitätsgrades stattfindet, keine Rekursion enthalten.

- Das Programm muß über sinnvolle Stellen für die Einstellung des Parallelitätsgrades verfügen. Abschnitt 5.3 erläutert die Punkte, auf die der Benutzer bei der Festlegung der Einstellungsstellen achten muß.

- Die Einstellung des Parallelitätsgrades muß für das Programm sinnvoll einsetzbar sein, d.h. eins der für die Einstellungsstrategien definierten Ziele (siehe Abschnitt 5.5.3) muß erreichbar sein.

Daraus ergeben sich für die Arbeit folgende Aufgaben:

- Strategien für eine effiziente Einstellung des Parallelitätsgrades müssen entwickelt werden. Diese Strategien entscheiden anhand von Laufzeitinformationen, ob die Anzahl der vom Programm belegten Prozessoren erhöht oder erniedrigt werden soll.

- Ein Mechanismus für die Datenumverteilung muß spezifiziert werden. Solch ein Mechanismus soll effizient und ausreichend generisch sein, damit unterschiedliche Datenstrukturen zwischen den allozierten Prozessoren umverteilt werden können, wenn eine Änderung des Parallelitätsgrades vorgenommen wird. Zudem sollen die Spezifizierung und Realisierung der Datenumverteilung neuer, vorher nicht betrachteter Datenstrukturen für den Benutzer möglichst leicht gemacht werden.

- Es soll ermöglicht werden, Messungen für die Einstellung des Parallelitätsgrades transparent für den Benutzer innerhalb einer Schleife im Programm durchzuführen. Diese Messungen liefern den Strategien die nötigen Parameter, beispielweise die Ausführungszeit oder die Effizienz der Schleife. Die Kosten der Messungen sollen gering gehalten werden, damit der durch die Einstellung insgesamt verursachte Mehraufwand klein bleibt.

- Es soll eine Bibliothek und deren Schnittstelle spezifiziert werden, über die der Benutzer die Stellen im Programm definiert, an denen die Einstellung stattfinden wird, und die Parameter für die Einstellung spezifiziert.

- Dem Benutzer muß es ermöglicht werden, die für die Einstellung benötigten Parameter zu spezifizieren (siehe Abbildung 5.2). Beispiele für diese Parameter sind die Strategie der Einstellung, die umzuverteilenden Strukturen und die Häufigkeit der Einstellung.

- Sowohl eine *Intra-Lauf-* als auch eine *Inter-Lauf*-Einstellung des Parallelitätsgrades sollen angeboten werden. Die beiden Möglichkeiten sollen dem Benutzer mittels der Bibliotheksroutinen zur Verfügung gestellt werden. Die erste Art der Einstellung findet innerhalb einer Ausführung des Programmes statt, während die zweite zwischen mehreren Läufen des Programmes geschieht. Die Inter-Lauf-Einstellung ist die Alternative, die das Modell für solche parallele Programme anbietet, bei denen die Einstellung des Parallelitätsgrades nur mit sehr hohen Kosten wegen der Datenumverteilung zu realisieren wäre.

Der Ablauf der Einstellung des Parallelitätsgrades ist in Abbildung 5.2 gezeigt. Das parallele Programm, dessen Parallelitätsgrad eingestellt werden soll, wird durch das Einfügen von Bibliotheksaufrufen und durch die Spezifizierung von Parametern vom Benutzer instrumentiert. Nach der Instrumentierung kann das Programm mit der Einstellung des Parallelitätsgrades ausgeführt werden.

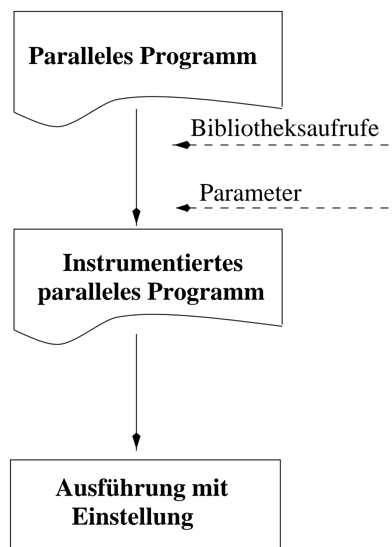


Abbildung 5.2: Überblick über den Ablauf der Einstellung des Parallelitätsgrades

5.2 Szenario der Einstellung des Parallelitätsgrades

Ein paralleles Programm, für das der Parallelitätsgrad eingestellt werden soll, wird als eine Menge parallelisierbarer Schleifen betrachtet, die unterschiedliche optimale Parallelitätsgrade haben können. Die betrachteten Schleifen wenden das parallele Programmiermodell *Datenparallelität* an (siehe Abschnitt 2.3), indem sie die gleichen Operationen auf unterschiedlichen Daten durchführen. Die Einstellung des Parallelitätsgrades kann in einer einzigen parallelisierbaren Schleife, aber auch in beliebig vielen Schleifen des Programmes stattfinden, z.B. an den Stellen a, b und c in Abbildung 5.3.

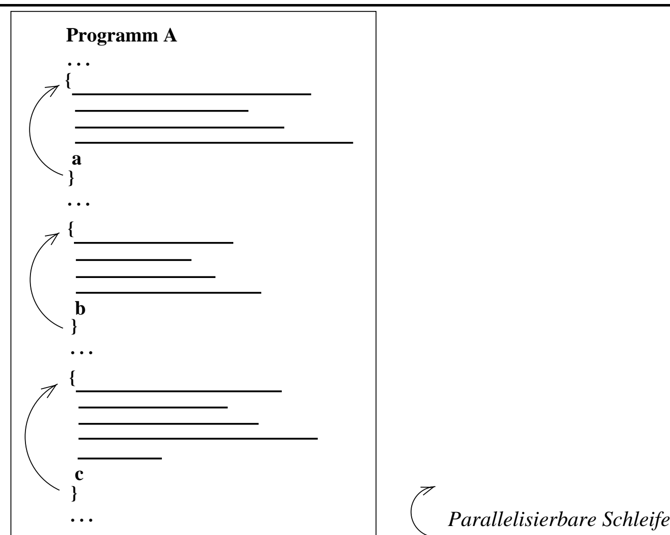


Abbildung 5.3: Stellen (a, b und c) der Einstellung des Parallelitätsgrades in einem Programm

Jede parallelisierbare Schleife, in der die Einstellung des Parallelitätsgrades stattfindet,

besitzt eine Einstellungsumgebung. Eine **Einstellungsumgebung** (*adaptation environment*) besteht aus Daten, die vom Benutzer spezifiziert und als Parameter für die Einstellung benutzt werden. Beispiele dazu sind die Häufigkeit der Einstellung, die bei Änderung des Parallelitätsgrades umzuverteilenden Datenstrukturen, die Virtualisierung[2] der Datenstrukturen, auf Basis deren die Problemgröße ausgerechnet wird, die Art der Datenverteilung (blockweise oder zyklisch), die Einstellungsstrategie und die von ihr benötigten Parameter, wie die Effizienz- bzw. die Beschleunigungsschranke.

Die Dynamik wird dabei folgendermaßen definiert: Wenn eine parallelisierbare Schleife, in der die Einstellung stattfinden wird, betreten wurde, wird die Einstellungsumgebung aktiviert. Wenn die Schleife zu Ende ist, wird der ursprüngliche Zustand, d.h. der Zustand vor der Einstellung, wiederhergestellt und die Einstellungsumgebung deaktiviert. Der ursprüngliche Zustand entspricht dem Zustand, bei dem die Prozessoranzahl vom Benutzer oder vom Betriebssystem für die Ausführung des Programmes festgelegt wurde, und wird beibehalten, bis eine weitere Schleife betreten wird, für die eine Einstellungsumgebung existiert.

5.3 Festlegung der Stellen für die Einstellung

Wie schon erwähnt, legt der Benutzer fest, wo der Parallelitätsgrades im Programm eingestellt werden soll. Beim Aussuchen dieser Stellen soll der Benutzer folgende Aspekte berücksichtigen:

- Wenn bei einer parallelisierbaren Schleife die Datenmenge, die bei einer Änderung des Parallelitätsgrades auf die Prozessoren umverteilt werden muß, sehr umfangreich ist, kann passieren, daß die Einstellung des Parallelitätsgrades keinen Gewinn oder sogar einen Verlust mit sich bringt. Die Höhe dieses Verlustes ist bedingt durch den Mehraufwand der Datenumverteilung. Noch kritischer wird diese Situation falls die parallelisierbare Schleife zu feingranular ist.[3]

- Wenn die Anzahl der Iterationen bei einer parallelisierbaren Schleife gering ist, wird sich die Einstellung des Parallelitätsgrades über ein Suchverfahren u.U. nicht lohnen. Der Grund dafür ist, daß die mit der Einstellung gebundenen Kosten sich innerhalb von wenigen Iterationen nur schwer amortisieren laßt. Bei diesem Fall passiert es auch, daß das Suchverfahren sehr früh abgebrochen werden muß. Dabei hat das Suchverfahren den optimalen Parallelitätsgrad evtl. noch nicht gefunden, wobei u.U. keine Aussage über den zuletzt eingestellten Parallelitätsgrad getroffen werden kann.

Das hier vorgeschlagene Modell sieht aber Alternativen für diese kritischen Fälle vor. In diesen Situationen kann evtl. eine Inter-Lauf-Einstellung (siehe Abschnitt 5.8) durchgeführt werden. Eine andere Alternative ist, den Parallelitätsgrad über ein mathematisches Modell (siehe Kapitel 4) zu bestimmen, falls der Berechnungs- und der Kommunikationsaufwand bekannt sind.

---

[2]Virtualisierung ist eine Technik, über die v logische (virtuelle) Prozessoren auf jedem der p physikalischen Prozessoren eines Parallelrechners bereitgestellt werden, wobei N unabhängige Datensätze auf die p Prozessoren ($v = N/p$) verteilt werden.

[3]Hier wird *Granularität* als Synonym für die Menge der Operationen benutzt, die es in einer parallelisierbaren Schleife pro Durchlauf zu berechnen gibt.

5.4 Suchverfahren zur Parallelitätsgradbestimmung

Der Parallelitätsgrad wird hier basierend auf einem Suchverfahren bestimmt, nämlich der Binärsuche [38]. Das Programm startet mit der maximalen Prozessoranzahl, die durch den Benutzer oder durch das Betriebssystem angegeben wurde. Der Parallelitätsgrad wird solange halbiert, bis das Optimum anhand einer Einstellungsstrategie (siehe Abschnitt 5.5) gefunden wird. Es handelt sich hier um eine vereinfachte Binärsuche, da nur Zweierpotenzen als Prozessoranzahl berücksichtigt werden. Der Aufwand bzw. die Zeit für das Finden des Optimums beträgt bei der Binärsuche $O(\log N)$, wobei N für die initiale Anzahl von Prozessoren steht.

Die Binärsuche, die als Ausgangspunkt die maximale Prozessoranzahl hat, ist besonders vorteilhaft für gutparallelisierbare Anwendungen. Weil solche Anwendungen genug Parallelismus anzubieten haben, befindet sich das Optimum mit einer hohen Wahrscheinlichkeit in der oberen Hälfte des Suchraumes.

Nachteil der oben beschriebenen vereinfachten Binärsuche ist, daß es sich beim gefundenen optimalen Parallelitätsgrad eigentlich um eine Annäherung an das Optimum handelt. Die Kosten, die bei der Suche nach dem absoluten Optimum entstehen würden (d.h. bei der Durchführung der kompletten Binärsuche), sind aber wegen der Datenumverteilung, die sich bei einer Änderung des Parallelitätsgrades notwendig macht, zu hoch und würden die Einstellung des Parallelitätsgrades auf Rechnern mit verteiltem Speicher unmöglich machen. Anhand einer Benchmark-Sammlung von 7 Programmen konnte in Kapitel 7 jedoch bewiesen werden, daß der Einsatz der vereinfachten Binärsuche keine Einschränkung für die Einstellung des Parallelitätsgrades bedeutet.

5.5 Strategien für die Einstellung des Parallelitätsgrades

Der Parallelitätsgrad wird nach bestimmten Strategien eingestellt, die u.a. die Entwicklung der Beschleunigung, der Effizienz und der Kosten-Nutzen-Relation als Kriterien beobachten. Die in dieser Arbeit untersuchten Strategien werden hier ausführlich beschrieben.

Zwei Arten von Strategien wurden untersucht, nämlich *lokale* und *globale* Strategien. Im folgenden werden zuerst die lokalen und nachher die globalen Strategien vorgestellt. In den Beschreibungen werden die unten aufgeführten Bezeichner verwendet, deren Werte der Benutzer des Systems bestimmt:

- Schranke EffThreshold: Effizienzschranke, deren Unterschreitung bei einigen Strategien als Kriterium für die Einstellung betrachtet wird.

- Schranke SpdThreshold: Beschleunigungsschranke, deren Unterschreitung bei einigen Strategien als Kriterium für die Einstellung betrachtet wird.

- Periodendauer bzw. Zyklus RUNLEN: Anzahl der Aufrufe auf Adaptionsroutinen. Nachdem diese Anzahl überschritten wurde, wird eine neue Meßphase (bei den globalen Strategien) ausgelöst.

Wie im Abschnitt 5.4 diskutiert, wenden die nach dem Optimum suchenden Strategien die Binärsuche an, bei der der Parallelitätsgrad solange halbiert wird, bis die jeweiligen Kriterien erfüllt werden.

5.5.1 Lokale Strategien

Die Strategien, die zu dieser Gruppe gehören, sind fähig, ein lokales Optimum für den Parallelitätsgrad nach den jeweils relevanten Kriterien zu finden. Daher können die Strategien nur den besten Parallelitätsgrad unter den analysierten finden, und es kann selbstverständlich passieren, daß das gefundene Optimum nur ein lokales Optimum oder eine Annäherung an das Optimum darstellt.

Es werden hier die folgenden lokalen Strategien betrachtet: **R***untime* **I***ncremental* **S***earch* **S***trategy* (RISS), **E***fficiency-driven* **I***ncremental* **S***earch* **S***trategy* (EISS), **S***tepwise* **L***ook with* **E***fficiency-threshold and* **V***ariable* **S***tep* (SLEVS), **C***ost*-**B***enefit-driven* **I***ncremental* **S***earch* **S***trategy* (CBISS) und **P***rediction-driven* **S***earch* **S***trategy* (PeSS). Die Strategien RISS, CBISS und PeSS wurden im Rahmen dieser Arbeit entwickelt. EISS und SLEVS basieren auf [44]. Sie gehören zu den *inkrementellen Testschritten* der *direkten Optimum-Suchverfahren* [27] und gehen von der intuitiven Idee aus, den Parallelitätsgrad schrittweise zu erhöhen, solange man dadurch eine Verbesserung des Leistungsverhaltens (d.h. der Beschleunigung, der Effizienz oder der Kosten-Nutzen-Relation) erreicht. Hier wird aber der Parallelitätsgrad erniedrigt, weil das Programm mit der maximalen Anzahl von Prozessoren gestartet wird. Das hat sich anhand einer Studie als angemessen erwiesen, besonders wenn der optimale Parallelitätsgrad sich weit von 1 befindet, weil durch diese Vorgehensweise Einstellungsschritte erspart bleiben, die auf sehr niedrigen Anzahl von Prozessoren stattfinden und deswegen für solche Programme nur mehr Zeit kosten. Verschlechtert sich das analysierte Leistungsverhalten infolge der Erniedrigung des Parallelitätsgrades, so wird die Suchrichtung umgekehrt, und der letzte Wert für den Parallelitätsgrad als Optimum angenommen.

- **R***untime* **I***ncremental* **S***earch* **S***trategy* (RISS)

  Bei dieser Strategie ist die Minimierung der Ausführungszeit der parallelisierbaren Schleife das Kriterium, wonach der Parallelitätsgrad eingestellt wird. Die Anzahl der zu belegenden Prozessoren wird solange halbiert, bis sich die Ausführungszeit nicht mehr verringert, sondern sich erhöht.

  Der Vorteil dieser Strategie ist, daß die Ausführungszeit der parallelisierbaren Schleife bei einem einzigen Prozessor nicht gemessen werden muß, weil die Werte der Beschleunigung und der Effizienz nicht benötigt werden. Der optimale Parallelitätsgrad wird dadurch schneller und deswegen mit geringeren Kosten erreicht, besonders wenn der optimale Parallelitätsgrad weit von 1 liegt. Die drei oben erläuterten Strategien benötigen dagegen die Messung mit einem Prozessor für die Berechnung der Beschleunigung bzw. der Effizienz.

- **E***fficiency-driven* **I***ncremental* **S***earch* **S***trategy* (EISS)

  Die Effizienz wird für die parallelisierbare Schleife gemessen. Wenn die Effizienz kleiner ist als die Effizienzschranke EffThreshold, wird die Prozessoranzahl halbiert. In anderen Fällen bleibt die Prozessoranzahl unverändert.

- **S***tepwise* **L***ook with* **E***fficiency-threshold and* **V***ariable* **S***tep* (SLEVS)

  Nach dieser Strategie wird die Prozessoranzahl solange erniedrigt, bis die berechnete Effizienz größer oder gleich der Effizienzschranke EffThreshold ist. Die steigende Tendenz kann durch die Beobachtung vorheriger Schritte festgestellt werden. Die Anzahl der zu betrachtenden vorherigen Schritte wird vom Benutzer durch die Einstellung des

Parameters *RUNLEN* festgelegt. Anhand dieser Tendenz wird versucht, den optimalen Grad schneller zu finden, indem die Schrittweite sukzessive erhöht wird.

Die Schrittweite der vorherigen Einstellungen bestimmen die aktuelle Schrittweite. Wenn zum Beispiel die drei letzten Einstellungen in der gleichen Richtung geschehen sind und die Effizienz sich unterhalb der Effizienzschranke befindet, kann die Schrittweite inkrementiert werden.

- **C***ost*-**B***enefit-driven* **I***ncremental* **S***earch* **S***trategy* (CBISS)

  Die Strategie CBISS sucht das Minimum der Kosten-Nutzen-Relation (siehe Abschnitt 2.4). Die Ausführungszeit und die Effizienz werden für die aktuelle Prozessoranzahl gemessen und die Kosten-Nutzen-Relation berechnet. Diese Strategie beantwortet in jedem ihrer Schritte die Frage, ob es sich eher lohnt, weniger Prozessoren einem Programm zuzuordnen. Wenn sich die Relation im Vergleich zu dem vorherigen Schritt reduziert hat, wird die Anzahl der Prozessoren halbiert. Andernfalls bleibt die Prozessoranzahl unverändert.

- **P***rediction-driven* **S***earch* **S***trategie* (PeSS)

  Das Ziel dieser Strategie ist die Minimierung der Ausführungszeit von Programmen. Dafür wird die Ausführungszeit der parallelisierbaren Schleife für den nächstmöglichen Parallelitätsgrad vorhergesagt. Die Ausführungszeitskurve eines parallelen Programmes bzw. eines parallelen Programmabschnittes gleicht der Form einer Badewanne, d.h. die Ausführungszeit sinkt bis zu einer gewissen Anzahl von Prozessoren ab und beginnt danach wieder zu steigen. Gesucht wird letztendlich der Wert für den Parallelitätsgrad, bei dem die Ausführungszeit der Schleife am geringsten ist (siehe Punkt K in Abbildung 5.4).[4] Die Kurve in Abbildung 5.4 soll „von rechts nach links“ betrachtet werden. Die Vorhersage der Ausführungszeit basiert auf zwei Annahmen:

  1. Die Ausführung eines Programmes mit p Prozessoren ist p mal schneller als mit einem einzigen Prozessor. Darauf muß noch die Kommunikationszeit addiert werden. Zusammengefaßt ergibt sich für die Bestimmung der Ausführungszeit mit p Prozessoren (T_p):

  $$\boxed{T_p = \frac{T_1}{p} + T_{comm}}$$

     p bezeichnet die Anzahl der Prozessoren und T_1 die Ausführungszeit mit einem Prozessor. D.h. die Abschätzung der Ausführungszeit mit p Prozessoren ist die Summe zweier Kurven, nämlich der für die Berechnungszeit und der für die Kommunikationszeit (siehe Abbildung 5.4).

  2. Die Kommunikationszeit läßt sich durch ein Polynom annähern:

  $$T_{comm} = c_2 \times p^2 + c_1 \times p + c_0$$

     c_2, c_1 und c_0 sind Konstanten, die durch eine Methode für die Lösung der linearen Gleichungen berechnet werden.

[4] K repräsentiert den Punkt, an dem zusätzliche Prozessoren zu einer Erhöhung der Ausführungszeit des Programmes führt. K wird auch Sättigungspunkt (*saturation point*) genannt.

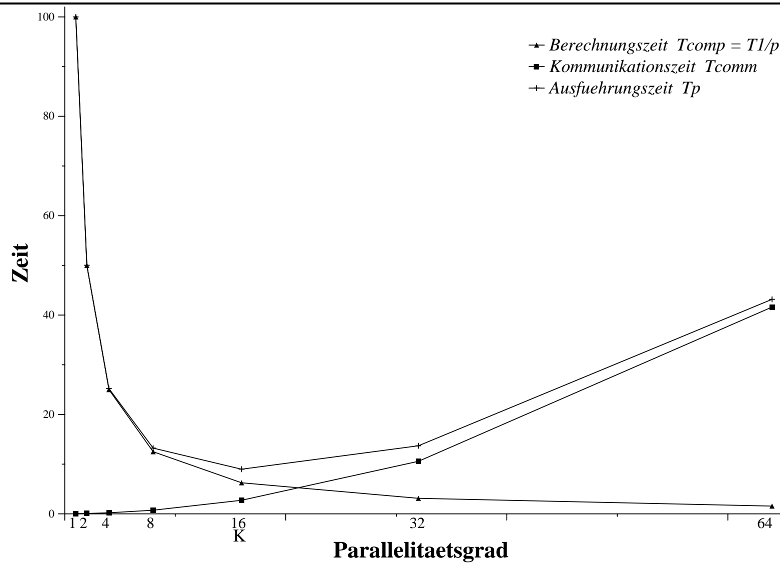


Abbildung 5.4: Kurven der Berechnungs-, Kommunikations- und Ausführungszeit für ein fiktives Programm

In der Gleichung

$$T_p = \frac{T_1}{p} + (c_2 \times p^2 + c_1 \times p + c_0) \tag{5.1}$$

sind T_1, c_2, c_1 und c_0 unbekannt. Es werden vier Meßpunkte betrachtet und die entsprechenden Gleichungen gebildet:

$$T_p = \frac{T_1}{p} + c_2 \times p^2 + c_1 \times p + c_0 \tag{5.2}$$

$$T_{\frac{p}{2}} = \frac{T_1}{\frac{p}{2}} + c_2 \times \left(\frac{p}{2}\right)^2 + c_1 \times \frac{p}{2} + c_0 \tag{5.3}$$

$$T_{\frac{p}{4}} = \frac{T_1}{\frac{p}{4}} + c_2 \times \left(\frac{p}{4}\right)^2 + c_1 \times \frac{p}{4} + c_0 \tag{5.4}$$

$$T_{\frac{p}{8}} = \frac{T_1}{\frac{p}{8}} + c_2 \times \left(\frac{p}{8}\right)^2 + c_1 \times \frac{p}{8} + c_0 \tag{5.5}$$

Die linearen Gleichungen werden durch die Gauss-Jordan-Eliminierungsmethode [5] gelöst und die Unbekannten berechnet. Danach wird der Wert für die Ausführungszeit am fünften Meßpunkt nicht gemessen, sondern durch die Gleichung (5.1) approximiert. Wenn dieser Wert niedriger ist als der vorherige, wird der Parallelitätsgrad halbiert. Andernfalls wird er nicht geändert. Falls die Halbierung vorgenommen wird, wird die Messung durchgeführt und der geschätzte Wert der Ausführungszeit durch den realen Wert korrigiert. Dies stellt im nächsten Schritt den neuen vierten Meßpunkt dar, der für die Schätzung der nächsten Ausführungszeit, d.h. mit halb so vielen Prozessoren, benutzt wird.

Die Anwendung dieser Strategie hat zwei Vorteile. Erstens muß die Ein-Prozessor-Messung nicht durchgeführt werden, weil die Werte der Beschleunigung sowie der Effizienz nicht benötigt werden. Der optimale Parallelitätsgrad wird dadurch schneller und deswegen auch mit geringeren Kosten gefunden. Zweitens geschieht die Datenumverteilung bzw. die Einstellung des Parallelitätsgrades nur dann, wenn es sich lohnt. D.h. erst nachdem abgeschätzt wurde, ob sich die Ausführungszeit mit wenigen Prozessoren voraussichtlich reduzieren wird, wird die Anzahl der belegten Prozessoren halbiert. Die Kosten für die Datenumverteilung bleiben also erspart, wenn sich die Änderung des Parallelitätsgrades für nicht lohnend geschätzt wurde.

Die zweite Annahme kann insofern geändert werden, indem die Kommunikationszeit durch ein Polynom erster Ordnung gegeben wird, falls eine lineare Kommunikationszeitkurve sich der Realität annähert. In dem Fall werden nur drei statt vier Meßpunkte für die Durchführung der Vorhersage gebraucht, was ein Kostenersparnis für die Einstellung des Parallelitätsgrades mit sich bringt. Die im Kapitel 7 präsentierten Ergebnisse wurden durch den Einsatz von Polynomen zweiter Ordnung erzielt.

Bei der Realisierung des entwickelten Modells wurden die Strategien RISS, EISS, CBISS und PeSS berücksichtigt.

5.5.2 Globale Strategien

Die Strategien zweiter Art können ein globales Optimum finden, indem alle möglichen Parallelitätsgrade analysiert werden, bevor der optimale Parallelitätsgrad ausgewählt und zur Programmausführung verwendet wird. Es werden hier die folgenden auf [44] basierenden globalen Strategien betrachtet: **E**_fficiency-driven_ **G**_lobal_ **S**_earch_ **S**_trategy_ (EGSS), **S**_peed-up-driven_ **G**_lobal_ **S**_earch_ **S**_trategy_ (SGSS) und **G**_lobal_ **S**_earch with_ **M**_aximum_ **S**_peed-up_ (GSMS).

Globale Strategien sehen eine Aufteilung der Einstellung des Parallelitätsgrades in drei Phasen vor. In der ersten Phase werden Meßwerte gesammelt. Dabei wird die Prozessoranzahl auf alle möglichen Werte, d.h. auf alle Potenzen von 2 innerhalb des durch die maximale Prozessoranzahl vorgegebenen Intervalls, gesetzt und das Programmverhalten gemessen. In der nächsten Phase findet die Bestimmung des optimalen Parallelitätsgrades nach den Kriterien der jeweiligen Strategie statt. In der dritten Phase wird der gewählte Parallelitätsgrad eingestellt und beibehalten, bis der Zyklus zu Ende geht. Die Länge des Zyklus wird durch den Parameter *RUNLEN* oder durch eine Änderung der Problemgröße bestimmt. Danach wird der Parallelitätsgrad auf die maximale Prozessoranzahl zurückgesetzt und wieder in die erste Phase übergangen.

Im folgenden werden die bei den verschiedenen Strategien eingesetzten Kriterien erläutert:

- **E**_fficiency-driven_ **G**_lobal_ **S**_earch_ **S**_trategy_ (EGSS)

  Die Kriterien bei dieser Strategie heißen: Effizienz größer als die Effizienzschranke EffThreshold und maximale Beschleunigung.

- **S**_peed-up-driven_ **G**_lobal_ **S**_earch_ **S**_trategy_ (SGSS)

  Für diese Strategie sind die folgenden Kriterien wichtig: Beschleunigung größer als die Beschleunigungsschranke SpdThreshold und maximale Effizienz.

- **G***lobal* **S***earch with* **M***aximum* **S***peed-up* (GSMS)

  Bei dieser globalen Strategie ist die maximale Beschleunigung als Kriterium vorgesehen.

Der Nachteil der globalen Strategien sind die dadurch entstehenden Kosten. Weil jeder Parallelitätsgrad analysiert wird, bevor der optimale ausgewählt werden kann, findet die Datenumverteilung sehr häufig statt. Außerdem dauert es entsprechend lange, bis das Programm mit einer geeigneten Anzahl von Prozessoren läuft. Darauf basierend werden die globalen Strategien bei der Realisierung des entwickelten Modells nicht weiterhin betrachtet.

5.5.3 Übersicht über die Strategien

Das Diagramm 5.5 gibt einen Überblick über die Strategien und die entsprechenden Ziele bei der Einstellung des Parallelitätsgrades. Die Strategien RISS, PeSS, SGSS und GSMS erzielen die Verbesserung des Laufzeitverhaltens eines Programmes. Die Strategie EISS und SLEVS hingegen unterstützen die Verbesserung der Auslastung des Gesamtsystems. Die Strategie CBISS hat die Kombination dieser zwei Aspekte als Ziel.

Strategie:		Ziel:
RISS, PeSS	→	Minimierung der Ausführungszeit
EISS, SLEVS	→	Einhaltung einer Effizienzschranke
CBISS	→	Minimierung der Kosten-Nutzen-Relation
EGSS	→	Einhaltung einer Effizienzschranke und Erreichen maximaler Beschleunigung
SGSS	→	Einhaltung einer Beschleunigungsschranke und Erreichen maximaler Effizienz
GSMS	→	Maximierung der Beschleunigung

Abbildung 5.5: Übersicht über die Strategien und ihre Ziele

5.6 Laufzeitinformationen und Messungen

Um den Strategien die benötigten Informationen für die Einstellung des Parallelitätsgrades zur Verfügung zu stellen, werden Messungen zur Laufzeit des parallelen Programmes durchgeführt. Die Ausführungszeit jeder Iteration der parallelisierbaren Schleife wird gemessen und den Strategien übergeben, wobei die Häufigkeit dieser Messung durch einen vom Benutzer einstellbaren Parameter definiert wird. Strategien wie EISS, SLEVS oder CBISS sowie alle globalen Strategien (siehe 5.5) brauchen zusätzlich die Ausführungszeit der Schleife mit einem Prozessor, damit sie die Effizienz, die Beschleunigung bzw. die Kosten-Nutzen-Relation berechnen können. Die eigentlichen Messung bzw. Berechnung dieser Größen verursachen einen geringen Mehraufwand, da sie nur Operationen wie Zeiterfassung und einfache Berechnungen benötigen.

Diese Messungen werden vollständig transparent für den Benutzer durchgeführt.

5.7 Datenumverteilung

Jedesmal, wenn der Parallelitätsgrad eines Programmes geändert wird, müssen die Daten zwischen den aktiven Prozessoren umverteilt werden, da das Modell einen Parallelrechner mit verteiltem Speicher vorsieht. Ein Prozessor ist *aktiv*, wenn er zu der Menge der Prozessoren gehört, die gerade an der Berechnung teilnehmen. Diese Menge entspricht den Prozessoren $P_0..P_{Parallelit\ddot{a}tsgrad-1}$ zu einem bestimmten Zeitpunkt. Hier wird der im Rahmen dieser Arbeit entwickelte Mechanismus zur Datenumverteilung vorgestellt.

Die Daten werden homogen über die aktiven Prozessoren verteilt, d.h. alle Prozessoren bekommen eine möglichst vergleichbare Menge von Datenumfang, da hier die Datenparallelität (siehe Abschnitt 2.3) ausgenutzt wird und eine gute Auslastung (*utilization*) der Prozessoren erwünscht ist.

Im folgenden werden die Grundidee bzw. die verschiedenen Arten der Datenumverteilung und die Strukturen, deren Umverteilung vom Modell unterstützt sind, präsentiert.

5.7.1 Grundidee

Wenn der Parallelitätsgrad sich ändert, werden die Daten umverteilt. Um eine Datenumverteilung durchzuführen, muß festgelegt werden, welche Prozessoren Daten exportieren (*Exporteur* oder *exporter*) bzw. welche importieren (*Importeur* oder *importer*). Abbildung 5.6 zeigt ein Beispiel, bei dem der Parallelitätsgrad von 4 auf 2 reduziert wird. In diesem Fall sind PE_1, PE_2 bzw. PE_3 Exporteure (abgehende Pfeile in Abbildung 5.6(b)) und PE_0 bzw. PE_1 Importeure (ankommende Pfeile in Abbildung 5.6(b)).

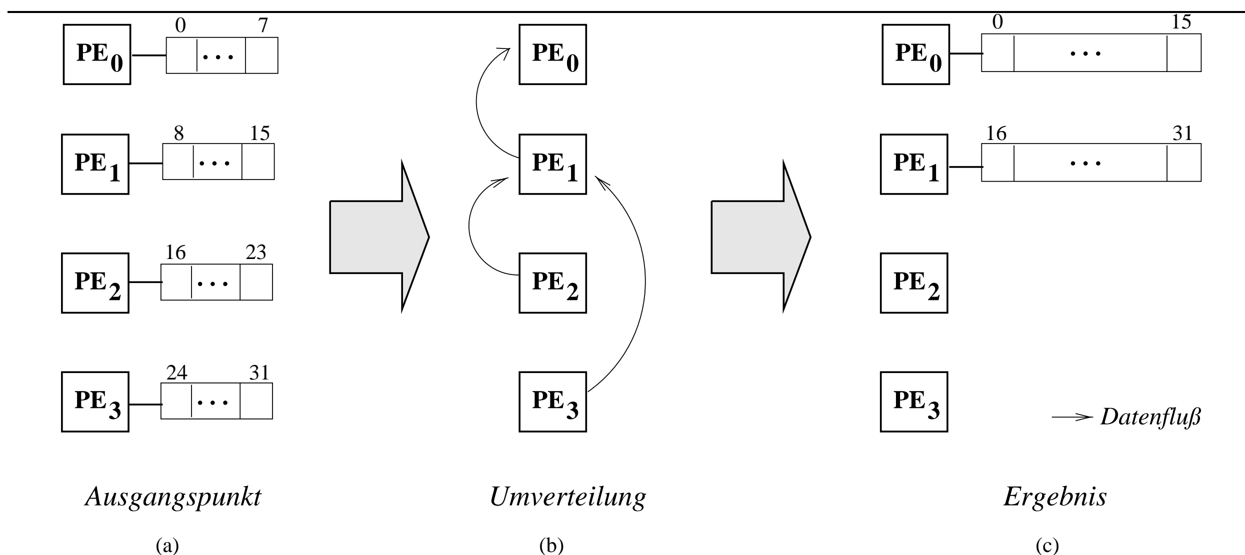


Abbildung 5.6: Datenumverteilung bei einer Änderung des Parallelitätsgrades

Der Zustand mit Parallelitätsgrad gleich 4 wird in Abbildung 5.6(a) und der resultierende Zustand mit Parallelitätsgrad gleich 2 in (c) mit den dazugehörigen Daten (Elemente eines Feldes) gezeigt, wobei der Virtualisierungsgrad bei einer Halbierung des Parallelitätsgrades verdoppelt wird. Hier werden die Elemente auf die Prozessoren blockweise verteilt.

Das Kommunikationsmuster ist in Abbildung 5.6(b) zu sehen. Die Zuordnung von Datenelementen zu Prozessoren wird durch die Formel $pe = global_index/Virt$ bei einer Blockverteilung (siehe Abschnitt 5.7.2) und durch die Formel $pe = global_index\%par_degree$ bei

einer zyklischen Verteilung (siehe Abschnitt 5.7.2) festgelegt, wobei *pe* die Prozessornummer, *global_ index* den globalen Index des entsprechenden Elementes, *Virt* den Virtualisierungsgrad und *par_ degree* den Parallelitätsgrad bezeichnen.[5] Die Zuordnung von Elementen zu Prozessoren variiert mit der Änderung des Parallelitätsgrades, aber diese Variation wird immer durch die oben beschriebenen Formel gegeben. Diese Methode hat drei Vorteile: Erstens bleibt der Mehraufwand der Berechnung für die neue Zuordnung von Datenelementen zu Prozessoren gering, zweitens ist die Berechnung der neuen Zuordnung dezentralisiert, d.h. jeder Prozessor kann anhand des Virtualisierungsgrades bzw. des Parallelitätsgrades selber die neue Zuordnung berechnen, so daß dabei keine Kommunikationskosten entstehen. Drittens wird kein Speicherplatz für aufwendige Transformationstabellen gebraucht.

5.7.2 Arten der Datenverteilung

Hier werden die zwei vom Modell unterstützten Arten der Datenverteilung beschrieben. Die Daten können blockweise oder zyklisch über die Prozessoren verteilt werden, je nachdem welche der beiden Arten zu der besten Datenlokalität für den parallelen Algorithmus führt. Die Art der Datenverteilung ist ein vom Benutzer anzugebender Parameter.

Blockweise Datenverteilung (*block data distribution*) – Bei dieser Art der Verteilung werden die Daten in Blöcken über die Prozessoren verteilt. Abbildung 5.7 zeigt ein Beispiel, bei dem acht Elemente eines Feldes über vier Prozessoren verteilt werden.

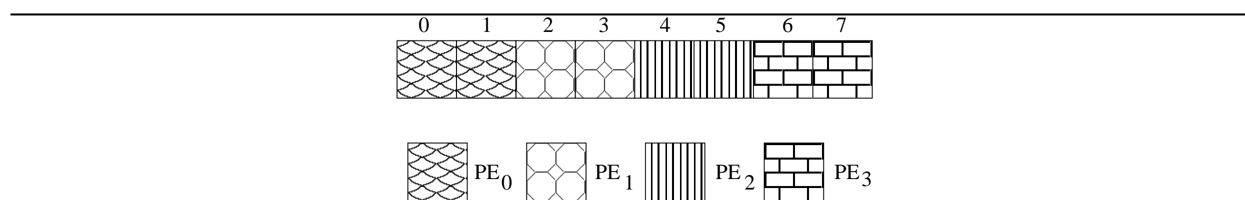


Abbildung 5.7: Beispiel einer Blockverteilung

Zyklische Datenverteilung (*cyclic data distribution*) – Bei dieser Art der Verteilung werden die Daten zyklisch über die Prozessoren verteilt. Abbildung 5.8 zeigt ein Beispiel, bei dem acht Elemente eines Feldes über vier Prozessoren zyklisch verteilt werden.

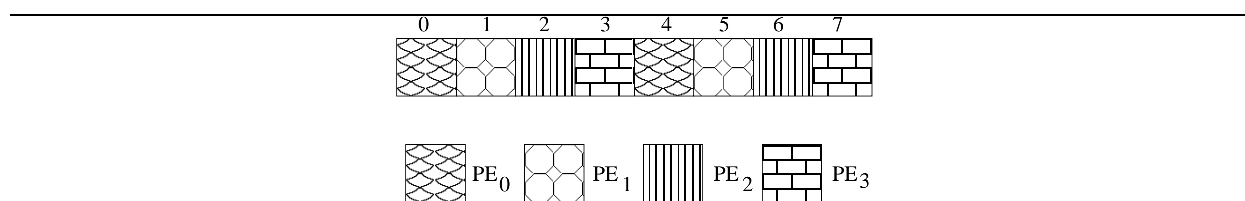


Abbildung 5.8: Beispiel einer zyklischen Blockverteilung

Die Datenumverteilung zwischen den Prozessoren unterstützt genau diese zwei Arten der Datenverteilung. Wenn der Benutzer aber ein anderes Verteilungsmuster (z.B. ein block-

[5] Das Symbol % repräsentiert die Modulo-Operation.

zyklisches Muster) bzw. eine heterogene Verteilung[6] anwenden möchte, kann er dies mit wenig Aufwand tun, indem er die Daten-Exportierung bzw. -Importierung spezifiziert (siehe Kapitel 6.5).

5.7.3 Unterstützte Datenstrukturen

Im Modell ist die Datenumverteilung für unterschiedliche Datenstrukturen vorgesehen, wobei ihre Datenelemente sowohl aus Gleitkomma- als auch aus Ganzzahlen bestehen können. Diese Datenstrukturen werden im folgenden beschrieben.

Feld (*array*) – eindimensionale Felder können blockweise oder zyklisch umverteilt werden.

Matrix (*matrix*) – Matrizen können blockweise oder zyklisch umverteilt werden. Die Umverteilung erfolgt außerdem zeilen- oder spaltenweise. Abbildung 5.9 zeigt zwei Beispiele: In (a) geschieht eine Blockverteilung zeilenweise und in (b) eine zyklische Verteilung spaltenweise.

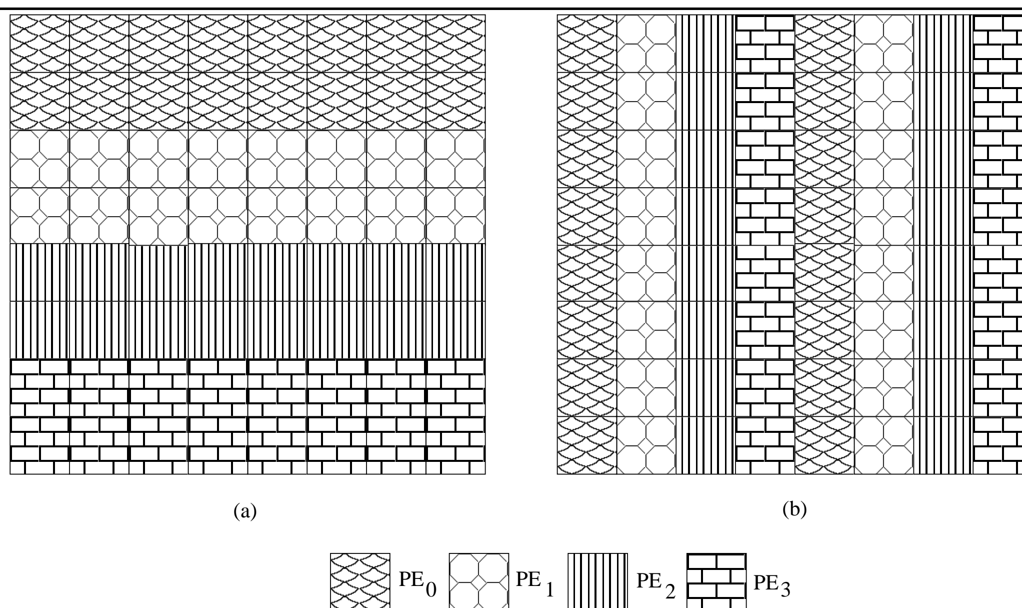


Abbildung 5.9: Verteilung einer Matrix; zeilen- und spaltenweise

Würfel (*cube*) – Analog zu den Matrizen können Würfel blockweise oder zyklisch, zeilen- bzw. spaltenweise umverteilt werden.

Strukturen (*structure*) – Felder von Verbünden können umverteilt werden, wobei die Verbünde von Feld zu Feld unterschiedlich sein können. Sie können unterschiedliche Attribute bzw. eine unterschiedliche Anzahl von Attributen enthalten.

[6] Eine heterogene Verteilung der Daten über die Prozessoren ist die Verteilung, bei der die Datenmenge der verschiedenen Prozessoren sehr unterschiedliche Größen haben können. Diese Art der Verteilung führt zu einer schlechten Auslastung der Prozessoren, weil einige Prozessoren sehr viel, andere dagegen sehr wenig zu berechnen haben.

Wenn der Benutzer aber andere Datenstrukturen im Modell einbetten möchte, kann er dies mit wenig Aufwand machen, indem er die dafür benötigte Initialisierung spezifiziert (siehe Abschnitt 6.5).

5.8 Arten der Einstellung

Der Parallelitätsgrad kann in zwei unterschiedlichen Weisen eingestellt werden: Als *intra-Lauf* oder *inter-Lauf*.

Intra-Lauf-Einstellung – Die Intra-Lauf-Einstellung findet innerhalb einer Ausführung des Programmes statt. Die Einstellung erfolgt schleifenweise, d.h. ggf. mehrmals innerhalb einer Ausführung des Programmes.

Inter-Lauf-Einstellung – Die Inter-Lauf-Einstellung findet zwischen verschiedenen Läufen des Programmes statt. Das Ziel ist, die Einstellung des Parallelitätsgrades auch bei solchen Programmen zu ermöglichen, für die eine Intra-Lauf-Einstellung wegen der Datenumverteilung nur mit sehr hohen Kosten zu realisieren wäre. Weil eine Änderung des Parallelitätsgrades bei der Inter-Lauf-Einstellung nur zwischen aufeinanderfolgenden Läufen des Programmes geschieht, macht sich eine Datenumverteilung nicht notwendig. Die Inter-Lauf-Einstellung wird so eingesetzt, daß der optimale Parallelitätsgrad vor der eigentlichen Ausführung des Programmes in sog. Vorläufen ermittelt wird. Das Programm wird so oft ausgeführt, bis der Parallelitätsgrad sich stabilisiert hat, d.h. bis zwei aufeinanderfolgende Ausführungen des Programmes den gleichen Parallelitätsgrad liefern. Nach diesen Vorläufen kann das Programm mit dem optimalen Parallelitätsgrad ausgeführt werden. Nach einer Veränderung der Problemgröße fängt die Einstellung erneut an.

Der Ablauf der Einstellung des Parallelitätsgrades ist in Abbildung 5.10 gezeigt. Sowohl für die Intra-Lauf- (siehe Abbildung 5.10(a)) als auch für die Inter-Lauf-Einstellung (siehe Abbildung 5.10(b)) wird zuerst überprüft, ob es für das Programm bereits einen Eintrag in einer Datenbank (siehe Abschnitt 5.9) gibt. Wenn dies der Fall ist, wird das Programm mit der registrierten Anzahl von Prozessoren ausgeführt. Am Anfang eines jeden Einstellungsschrittes wird ebenfalls überprüft, ob sich die Problemgröße geändert hat. Bei einer Änderung der Problemgröße wird die Einstellung erneut angefangen. Wenn kein entsprechender Eintrag in der Datenbank vorhanden ist, finden für die *Intra-Lauf*-Einstellung Messungen (siehe Abschnitt 5.6) innerhalb der Schleife statt. Als nächstes wird der Parallelitätsgrad nach der jeweils ausgewählten Strategie eingestellt und die Datenumverteilung durchgeführt. Falls sich der Parallelitätsgrad stabilisiert hat bzw. das Optimum gefunden wurde, wird der optimale Parallelitätsgrad in die Datenbank (siehe Abschnitt 5.9) aufgenommen. Andernfalls finden die Messungen nochmals statt.

Für eine *Inter-Lauf*-Einstellung wird das Programm dagegen mehrmals hintereinander ausgeführt, wobei die Messungen und die Adaption des Parallelitätsgrades nach der ausgewählten Strategie am Ende jeder Iteration stattfinden. Nachdem sich der Parallelitätsgrad stabilisiert hat, wird der gefundene optimale Parallelitätsgrad in die Datenbank eingetragen.

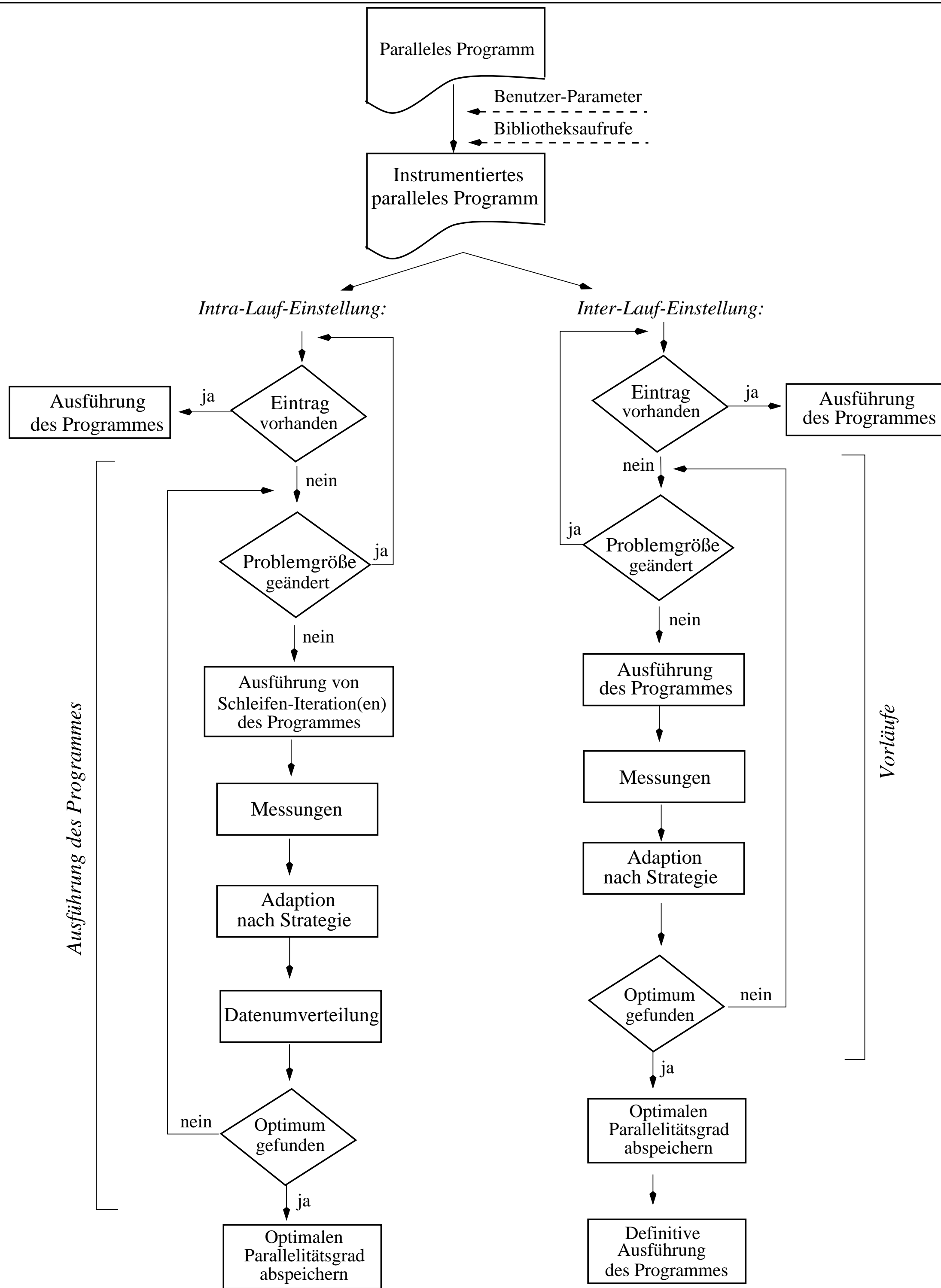


Abbildung 5.10: Überblick über den Ablauf der Einstellung des Parallelitätsgrades

5.9 Parallelitätsgrad-Datenbank

Jede zu Ende ausgeführte Einstellung des Parallelitätsgrades generiert einen Eintrag in eine Datenbank. Es handelt sich um die Parallelitätsgrad-Datenbank, die hier vorgestellt wird.

Für jedes paralleles Programm bzw. jede parallelisierbare Schleife, wofür der optimale Parallelitätsgrad einmal gefunden wurde, wird dieser mit dem Ziel aufgenommen, das Programm bzw. jede parallelisierbare Schleife im Programm bereits am Anfang der nächsten Ausführung mit der optimalen Anzahl von Prozessoren laufen lassen zu können. Beim Vorhandensein dieser Information muß der Parallelitätsgrad nur einmal am Anfang des Programmes bzw. der parallelisierbaren Schleifen eingestellt werden, falls die anfängliche Prozessoranzahl vom gefundenen Optimum überhaupt abweicht. Vorteil daran ist, daß das Programm so früh wie möglich mit der optimalen Anzahl von Prozessoren läuft und daß die Kosten für die Datenumverteilung im Fall einer Intra-Lauf-Einstellung sehr gering bleiben, da eine Änderungen des Parallelitätsgrades viel seltener vorkommt. Falls eine parallelisierbare Schleife, bei der eine Einstellung stattfindet, sich um keine innere Schleife einer Verschachtelung handelt, geschieht die Änderungen des Parallelitätsgrades sogar nur ein einziges mal.

Um überprüfen zu können, ob ein passender Eintrag für ein bestimmtes Programm in der Parallelitätsgrad-Datenbank vorhanden ist, müssen einige Daten zur Verfügung stehen. Ein Eintrag in der Datenbank besteht aus den Daten, die in Abbildung 5.11 zu sehen sind. Für jede Schleife im Programm, deren Parallelitätsgrad eingestellt wurde, ist ein Eintrag in der Datenbank vorgesehen.

Name	Dateidatum	MaxProz	SchleifeID	Strategie	Problemgröße	Parallelitätsgrad

Abbildung 5.11: Felder eines Parallelitätsgrad-Datenbank-Eintrages

Die verschiedenen Felder in Abbildung 5.11 werden im folgenden erläutert:

- Name – Spezifiziert den Dateinamen des Programmes, dessen Parallelitätsgrad eingestellt wurde.
- Dateidatum – Spezifiziert das Datum der Quelldatei, an dem die letzte Modifikation stattgefunden hat. Anhand dieser Information kann überprüft werden, ob Änderungen im Programm im Vergleich zu den vorherigen Einstellungen des Parallelitätsgrades vorgenommen wurden. Im Fall einer Änderung ist der evtl. vorhandene Parallelitätsgrad nicht mehr aktuell.
- MaxProz – Spezifiziert die maximal verfügbare Anzahl von Prozessoren bei der Ausführung des Programmes. Diese Information ist notwendig, weil bei unterschiedlichen maximalen Prozessoranzahlen unterschiedliche optimale Parallelitätsgrade gefunden werden können.
- SchleifeID – Identifiziert die Schleife im Programm, zu der der gefundene optimale Parallelitätsgrad gehört.
- Strategie – Spezifiziert die benutzte Strategie bei der Einstellung des Parallelitätsgrades.

- Problemgröße – Spezifiziert die Problemgröße für die Schleife, bei der der Parallelitätsgrad eingestellt wurde.

- Parallelitätsgrad – Spezifiziert den optimalen Parallelitätsgrad einer parallelisierbaren Schleife unter der angegebenen Strategie.

Nur wenn die sechs ersten Felder eines Datenbank-Eintrages mit den Informationen des aktuellen Programmes übereinstimmen, kann der gespeicherte Parallelitätsgrad für die aktuelle Einstellung übernommen werden.

Kapitel 6

Realisierung

Dieses Kapitel beschreibt die Realisierung der im Kapitel 5 präsentierten automatischen Einstellung des Parallelitätsgrades. Ein System namens *ParGrad*[1] wurde entwickelt, um das vorgeschlagene Modell zu bewerten und die Untersuchung paralleler Programme auf die Einstellung des Parallelitätsgrades hin zu ermöglichen.

Im folgenden werden das Szenario der Einstellung des Parallelitätsgrades und die angewendeten Werkzeuge präsentiert, sowie die Prozeßarchitektur und die Schnittstelle zwischen Benutzer und System beschrieben. Danach wird auf die Realisierung der Instrumentierung, der Strategien und der Datenumverteilung eingegangen. Die Voraussetzungen und Einschränkungen des Systems werden anschließend diskutiert.

6.1 Szenario der Einstellung des Parallelitätsgrades

In diesem Abschnitt wird die Anwendung des Systems zur Einstellung des Parallelitätsgrades von Programmen aus der Sicht des Benutzers diskutiert. Das System ist für die Einstellung des Parallelitätsgrades in C++ geschriebener paralleler Programme ausgelegt.[2] Wie in Abbildung 6.1 gezeigt wird, definiert der Benutzer das Ziel, das bei der Einstellung des Parallelitätsgrades verfolgt werden soll, nämlich die Verbesserung des Laufzeitverhaltens des parallelen Programmes, die Verbesserung der Systemauslastung oder ein Kompromiß dieser beiden Ziele. Anhand dieser Information wählt der Benutzer eine Strategie für die Einstellung des Parallelitätsgrades aus und spezifiziert die für die Einstellung benötigten Parameter (siehe Abschnitt 6.7.2). Außerdem soll der Benutzer als Autor des Programmes in der Lage sein, die Stellen im Programm zu benennen, an denen der Parallelitätsgrad sinnvollerweise eingestellt werden soll (siehe Abschnitt 6.7.1). Durch die Spezifizierung der Parameter bzw. der Stellen für die Einstellung erzeugt der Benutzer ein instrumentiertes paralleles Programm, das zur Einstellung des Parallelitätsgrades ausgeführt werden kann.

6.2 Einschränkungen

Die Einschränkungen des ParGrad-Systems werden im folgenden diskutiert:

- Das parallele Programm muß in der Programmiersprache C++ geschrieben sein.

[1] *ParGrad* steht für **Par**allelitäts**Grad**.

[2] Zur Einstellung des Parallelitätsgrades können in der Programmiersprache C geschriebene Programme sehr leicht auf C++ umkodiert werden, indem eine einzige Klasse spezifiziert wird und die im Programm existierenden Routinen und Funktionen als Methoden dieser Klasse geschrieben werden.

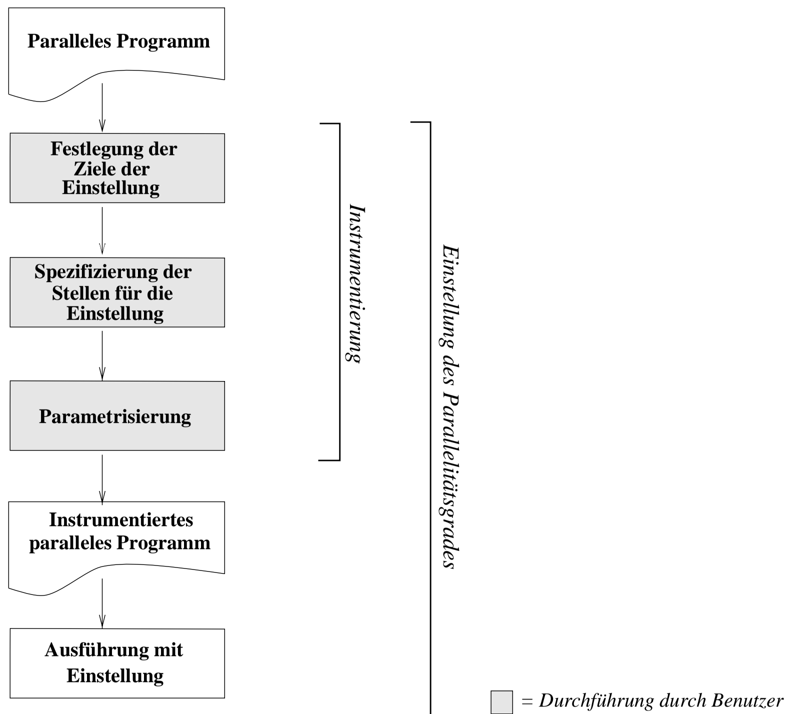


Abbildung 6.1: Übersicht über die Einstellung des Parallelitätsgrades

- Durch eine Inter-Lauf-Einstellung kann der Parallelitätsgrad entweder von einer einzigen parallelen Schleife oder vom gesamten Programm eingestellt werden. Es besteht nicht die Möglichkeit, über das System ParGrad den Parallelitätsgrad mehrerer paralleler Schleifen im Benutzerprogramm einzustellen, falls eine Inter-Lauf-Einstellung eingesetzt wird.

- Innerhalb einer Einstellungsumgebung müssen alle Datenstrukturen, die bei einer Änderung am Parallelitätsgrad umverteilt werden müssen, vom gleichen Typ sein. Die Art der Datenverteilung ist dabei einheitlich.

- Die Einstellung des Parallelitätsgrades findet für solche Programme statt, deren Problemgröße sich innerhalb der parallelisierbaren Schleife nicht ändert.

Die Programme der eingesetzten Benchmark-Sammlung (siehe Abschnitt 7.2.2) wurde von diesen Einschränkungen nicht betroffen.

6.3 Verwendete Werkzeuge

Das System *ParGrad* wurde unter einem auf UNIX basierenden Betriebssystem entwickelt, nämlich dem UNICOS/mk [41] Version 2.0.4.43. Das Betriebssystem UNICOS der Firma Cray Research basiert auf UNIX System V von UNIX System Laboratories und stellt das erste UNIX-basierte Betriebssystem für einen Superrechner dar. UNICOS erfüllt vollständig den X/Open XPG4 Base 95 Profile, einer Obermenge der POSIX 1003.1 und POSIX 1003.2 Standards. *mk* steht für *Microkernel* und deutet darauf hin, daß das UNICOS/mk auf einer *Microkernel*-Technologie basiert.

Als Werkzeug diente der Übersetzer Cray C++, der das ISO/ANSI *Draft Working Paper* für C++ unterstützt, wobei nicht der vollständige Sprachumfang implementiert ist.

Für die Kommunikation zwischen Prozessoren wurde die Cray-*shmem*-Bibliothek (*libsma*) eingesetzt. Die Routinen dieser Bibliothek unterstützen den logisch gemeinsamen, physikalisch verteilten Speicherzugriff, wobei sie den Mehraufwand der Nachrichtenkopplung bzw. die Latenzzeit minimieren und die Bandbreite maximieren.[3] Für die Kommunikation zwischen Prozessoren wurde zusätzlich die E-Register-Bibliothek [40] benutzt. Die E-Register sind ein externer Registersatz, über den der Zugriff auf entfernte Speicher behandelt wird.[4] Die in der Programmiersprache C geschriebenen Routinen dieser Bibliothek sprechen die E-Register der Cray direkt an und bringen dadurch für Datenmenge bis 128 Elemente eine höhere Leistung als die Cray-shmem-Routinen. Ab dieser Elementenzahl sind wiederum die shmem-Routinen effizienter [40].

6.4 Prozeßarchitektur

Die *ParGrad*-Prozeßarchitektur ist in Abbildung 6.2 dargestellt. Jedes an der Ausführung des Programmes beteiligte Prozessorelement (PE) führt genau einen ParGrad-Prozeß aus. Ein ParGrad-Prozeß besteht aus fünf Modulen, die im folgenden mit Hilfe von Abbildung 6.2 beschrieben werden.

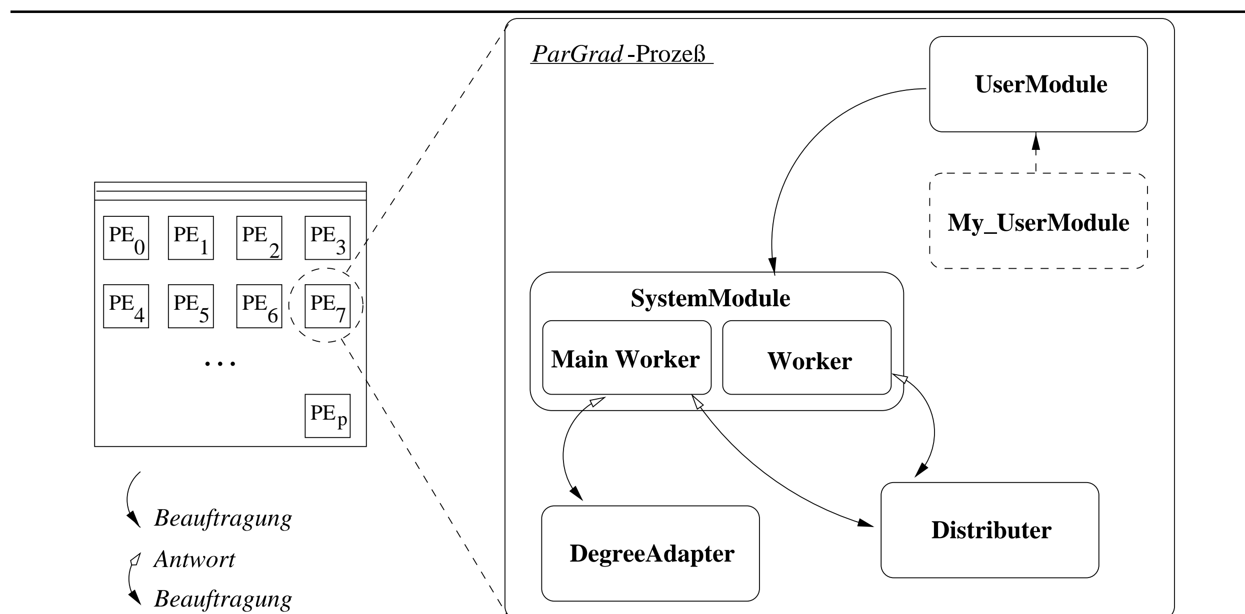


Abbildung 6.2: Prozeßarchitektur des *ParGrad*-Systems

My_UserModule – repräsentiert das Benutzerprogramm, dessen Parallelitätsgrad eingestellt werden soll.

[3]Latenzzeit ist die Zeit zwischen dem Starten des Datentransfers durch einen PE (Prozessorelement) und der Verfügbarkeit der Daten auf einem entfernten PE.

[4]E-Register sind zwar Teil des Adreßraumes (*memory mapped*), befinden sich jedoch in schnellen SRAM Chips.

UserModule – Dieses Modul repräsentiert die Schnittstelle zwischen ParGrad und dem Benutzerprogramm (*My_ UserModule*). Hier sind alle Methoden enthalten, über die der Benutzer von seinem Programm aus mit dem ParGrad-System interagieren kann. Diese Methoden werden im Abschnitt 6.7 ausführlich erläutert.

SystemModule – Dieses Modul repräsentiert den Verwalter der Einstellung des Parallelitätsgrades. Hier sind alle Methoden enthalten, die die Verwaltung der Einstellung implementieren, wie z.B. das Starten einer Datenumverteilung oder der konkreten Einstellung durch den Aufruf der Einstellungsstrategie. Auf diese Methoden wird in Abschnitt 6.8.2 näher eingegangen. Wie Abbildung 6.2 zeigt, besteht das *SystemModule* entweder aus einem *Main Worker* (Hauptarbeiter) oder aus einem normalen *Worker* (Arbeiter). Es existiert ein einziger *Main Worker* im gesamten System, wobei er auf dem logischen PE_0 plaziert ist. Die Anzahl der *Worker* entspricht dem zum Anfang der Ausführung des Programmes eingestellten Parallelitätsgrad, wobei sie auf PE_1 bis $PE_{Parallelitätsgrad-1}$ verteilt sind. Der Unterschied zwischen dem *Main Worker* und einem normalen *Worker* ist, daß der erstgenannte zusätzlich zu der Durchführung normaler Berechnungen fähig ist, eine konkrete Einstellung zu starten, d.h. er kann die Einstellung nach der angegebenen Strategie auslösen, und den neuen Parallelitätsgrad den anderen *Worker* mitteilen. Die konkrete Einstellung des Parallelitätsgrades ist auf dem *Main Worker* zentralisiert, da sie nur einmal stattfinden muß. Sowohl der *Main Worker* als auch ein *Worker* sind an einer Datenumverteilung beteiligt und können deshalb mit dem Modul *Distributer* interagieren.

DegreeAdapter – Dieses Modul ist für die konkrete Einstellung des Parallelitätsgrades nach der angegebenen Einstellungsstrategie zuständig. Es enthält alle Methoden, die die einzelnen Strategien implementieren. Diese Methoden werden im Abschnitt 6.9 ausführlich diskutiert. Der *DegreeAdapter* kann nur vom *SystemModule* des *Main Worker* aus aktiviert werden.

Distributer – Dieses Modul ist für die Datenumverteilung zuständig und enthält die dafür benötigten Methoden. Eine Datenumverteilung wird vom *Main Worker* gestartet, wobei alle aktiven *Worker* an der Datenumverteilung teilnehmen.[5] In dem Fall einer Inter-lauf-Einstellung (siehe Abschnitt 5.8) existiert das *Distributer*-Modul nicht in der Prozeßarchitektur.

6.5 Klassendiagramm

Das ParGrad-System wurde objektorientiert modelliert und implementiert. Die Abbildung 6.3 zeigt das *UML*-Klassendiagramm des ParGrad-Systems.[6] Im folgenden wird die Funktionalität der einzelnen Klassen erläutert.

Die Funktionalitäten der Klassen *My_ UserModule*, *UserModule*, *SystemModule*, *Main Worker*, *Worker*, *DegreeAdapter* und *Distributer* entsprechen der Beschreibung in Abschnitt 6.4 und werden hier nicht mehr diskutiert. Die anderen Klassen sind:

[5] Aktiv sind alle seit Anfang der Ausführung des Programmes gestarteten *Worker*, die an der Ausführung des Programmes gerade teilnehmen. Das entspricht den *Worker* auf PE_0 bis $PE_{Parallelitätsgrad-1}$.

[6] UML steht für *Unified Modeling Language* und stellt einen Standard für die Beschreibung objektorientierter Modelle dar.

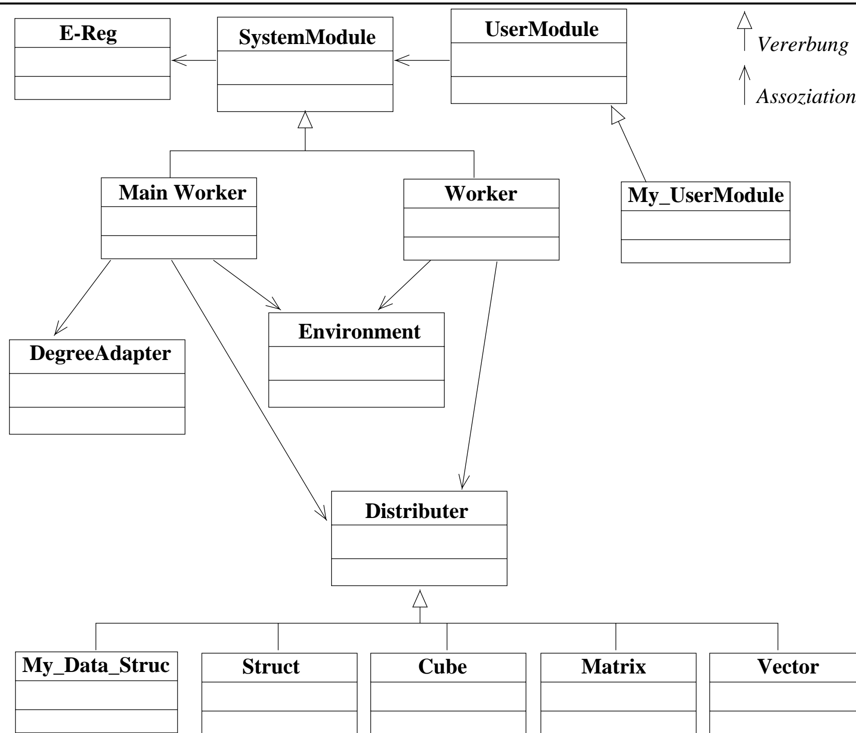


Abbildung 6.3: UML-Klassendiagramm des ParGrad-Systems

- **E-Reg** – Diese Klasse ist für die E-Register-Operationen (siehe Abschnitt 6.3) bzw. für den Zugriff auf die E-Register-Bibliothek [40] zuständig. Die Klasse *SystemModule* benutzt die E-Reg-Klasse.

- **Environment** – Definiert eine Einstellungsumgebung (*adaptation environment*; siehe Abschnitt 5.2) und ihre Komponenten. Objekte dieser Klasse werden durch die *SystemModule*-Subklassen *Main Worker* und *Worker* erzeugt. Ein ParGrad-Prozeß hat ein Objekt der Klasse *Environment* für jede einzelne Einstellungsumgebung.

- **Vector, Matrix, Cube, Struct** – Diese sind Subklassen der *Distributer*-Klasse und werden von den Klassen *Main Worker* bzw. *Worker* benutzt. Sie sind für die Datenumverteilung der entsprechenden Strukturen zuständig. Je nach Typ der Datenstrukturen (siehe Abschnitt 5.7.3), die nach einer Änderung des Parallelitätsgrades umverteilt werden müßen, wird ein Objekt der entsprechenden Klasse, also *Vector*, *Matrix*, *Cube* oder *Struct*, bei der Erzeugung einer neuen Einstellungsumgebung durch den *SystemModule* (d.h. durch den *Main Worker* oder den *Worker*, je nach ParGrad-Prozeß) kreiert.

- **My_DataStruc** – Diese Subklasse der *Distributer*-Klasse repräsentiert die Klassen, die durch den Benutzer des ParGrad-Systems spezifiziert werden können, falls er eine neue, im System nicht vorhandene Datenstruktur umzuverteilen hat. Der Benutzer spezifiziert die Datenstrukturen und die Methoden, die den Daten-Export bzw. -Import (siehe Abschnitt 5.7.1) implementieren.

6.6 Schnittstelle zwischen Benutzer und System

Das System ParGrad wurde als Bibliothek realisiert. Die Routinen dieser Bibliothek sind das einzige Mittel, wodurch der Benutzer mit dem ParGrad-System interagieren kann. Die Methoden eines Objektes der Klasse *UserModule* in Abbildung 6.3 entsprechen den Bibliotheksroutinen, die dem Benutzer zur Verfügung stehen.

Die Routine *ParGrad()* startet die Einstellung des Parallelitätsgrades in einer parallelen Schleife. Sie wird im Abschnitt 6.7.1 ausführlich diskutiert.

Die weiteren zur Verfügung stehenden Routinen werden in drei Gruppen klassifiziert, nämlich *set*-, *get*- und *miscellaneous*-Routinen. Im folgenden wird auf die Gruppen näher eingegangen.

- *set*-Gruppe: Die Routinen dieser Gruppe sind für das Setzen der verschiedenen Parameter einer Einstellungsumgebung zuständig. Beispiele solcher Parameter sind die umzuverteilenden Datenstrukturen, der Virtualisierungsgrad, die Art der Einstellung und die Strategie für die Einstellung.

- *get*-Gruppe: Mittels dieser Routinen kann der Benutzer das ParGrad-System die jeweils aktuellen Werte verschiedener Parameter abfragen. Parameter wie z.B. der Parallelitätsgrad, die Häufigkeit der Einstellung, der Virtualisierungsgrad verschiedener Datenstrukturen und die Einstellungsstrategie können abgefragt werden. Geliefert werden die Werte für die aktuellen Einstellungsumgebung.

- *Miscellaneous*-Gruppe: Diese Gruppe enthält Routinen, die allgemein im Benutzerprogramm zu gebrauchen sind. Beispiele dafür sind die Konvertierung von Elementenindizes von lokal auf global (*local2global(local_index)*) und auch umgekehrt (*global2local(global_index)*) sowie die Lokalisierung eines Elementes (d.h. in welchem PE es sich befindet) einer Datenstruktur über seinen Index (*index2PE(global_index)*). Für die Ausführung aller in Klammern angegebenen Aufrufen werden nur noch die entsprechenden Indizes gebraucht, weil sowohl der Virtualisierungsgrad als auch der Parallelitätsgrad und die Art der Datenverteilung, die für die Konvertierung der Indizes bzw. die Festlegung des betroffenen Prozessors notwendig sind, durch das ParGrad-System bekannt sind.

6.7 Instrumentierung

Wie im Abschnitt 6.1 bereits erwähnt wurde, legt der Benutzer die Stellen im Programm fest, wo die Einstellung des Parallelitätsgrades stattfinden soll. Außerdem spezifiziert er eine Reihe von Parametern, die für die automatische Einstellung benötigt werden. Sowohl die Festlegung der Stellen für die Einstellung als auch die Spezifizierung der Parameter geschehen mittels einer Instrumentierung des Benutzerprogrammes (siehe Abbildung 6.1). Im folgenden werden diese zwei Aktivitäten genauer beschrieben. Danach wird auf den Mehraufwand der Instrumentierung näher eingegangen.

Bestandteil der Instrumentierung ist auch die Deklaration des Benutzerprogrammes als Subklasse der ParGrad-Klasse namens *UserModule*, wie in dem folgenden Beispiel in der Zeile 1 gezeigt ist.

```
1    class My_UserModule: public UserModule
2    {
```

```
3    private:
4              int  get_log2_keys(int num_keys);
5              void all_fill_buckets(int pass);
6              void all_scan_buckets(int pass);
7              void all_coalesce(int pass);
8              ...
9              My_UserModule();
10             ~My_UserModule();
11
12   protected:
13               int *g_keys0;
14               int *g_keys1;
15               int *keys;
16  };
```

Ein Objekt namens *my_ module*, das zur Klasse *My_ UserModule* gehört, kann der Benutzer durch die Operation

$$My_UserModule \quad my_module;$$

erzeugen. Durch diese Operation wird der Klassenkonstruktor *My_ UserModule()* ausgeführt. Die Ausführung dieses Konstruktors spielt eine wichtige Rolle, wie in Abschnitt 6.7.2 erklärt wird.

6.7.1 Spezifizierung der Stellen für die Einstellung

Die Einstellung des Parallelitätsgrades findet auf Schleifenebene statt (siehe Abschnitt 5.2). Daher muß der Benutzer bei der Instrumentierung seines Programmes spezifizieren, in welcher paralleler Schleife die Einstellung stattfinden soll. Die Anzahl der Schleifen, in denen der Parallelitätsgrad eingestellt wird, kann beliebig sein.

Der Benutzer spezifiziert die Stellen für die Einstellung durch das Einsetzen von Aufrufen einer Bibliotheksroutine, nämlich die Routine *ParGrad()*, die eine Methode der Klasse *UserModule* ist. Dieser Aufruf wird an das Ende der ausgewählten Schleife plaziert, weil die Laufzeitmessungen, worauf die Einstellung des Parallelitätsgrades basiert, immer am Ende einer Iteration der Schleife geschehen sollen. Falls es mehrere Einstellungsstellen im Programm gibt, sieht ein Aufruf der Methode folgendermaßen aus: $ParGrad(Einstellungsumgebung)$. Der Parameter ist ein Bezeichner für die *Einstellungsumgebung*, da es in dem Fall mehrere davon gibt. Ein Beispiel dafür ist im folgenden anhand eines Code-Ausschnittes des Radix-Sortieralgorithmus (siehe Abschnitt 7.1.1) gezeigt, wobei der Aufruf der Einstellung in der Zeile 11 zu sehen ist:

```
1   for (pass = 0 ; pass < DIGITS_PER_KEY ; pass++) /* for each digit of a key */
2   {
3    all_fill_buckets(pass);  /* counts the number of keys with each digit on
4                                each processor */
5    all_scan_buckets(pass);  /* uses the local buckets histograms to determine
6                                the global key list offset */
7    all_coalesce(pass); /* moves all keys of a processor to their new locations
8                           via the offset array */
```

```
9
10     ...
11     ParGrad();
12   }
```

6.7.2 Spezifizierung der Parameter

Hier werden die Parameter präsentiert, die vom Benutzer bei der Instrumentierung eines parallelen Programmes angegeben werden. Im folgenden ist ein Beispiel zur Spezifizierung von Parametern für die Intra-Lauf-Einstellung eines Programmes, für das eine einzige Einstellungsumgebung existiert, gezeigt:

```
My_UserModule::My_UserModule()
{
 set(ENVIRONMENT, 0);
 set(NUMSOURCE, 1);
 set(NUMPAR, 1);
 set(DATA_DISTRIBUTION, BLOCK);
 set(DATA_STRUCTURE, VECTOR);
 set(PERIOD, 1);
 set(VIRT, KEYS_PER_PROC);
 set(SOURCE, 0, &keys);
 set(PAR, 0, &pass);
 set(PAR_VALUE, 0, DIGITS_PER_KEY - 1);
 set(STRATEGY, EISS);
 set(EFFTHRESHOLD, 0.50);
 set(FILENAME, "radix");
}
```

Der Benutzer spezifiziert im Konstruktor der Klasse *My_ UserModule* einen Parameterblock für jede Einstellungsumgebung, d.h. für jede Schleife, in der eine Einstellung des Parallelitätsgrades stattfinden soll. Um den Parameter zu spezifizieren, wendet er eine Methode der *set*-Gruppe (siehe Abschnitt 6.6) an, die zu der ParGrad-Bibliothek gehört. Zu einem Parameterblock gehört die folgende Parameter-Spezifizierung:

set(ENVIRONMENT, int environment_id) – Definiert den Bezeicher für die entsprechende Einstellungsumgebung. Falls dieser Parameter nicht spezifiziert wird, wird er auf Null initialisiert, d.h. daß es sich um die erste Einstellungsumgebung handelt.

set(NUMSOURCE, int source_num) – Legt die Anzahl der Datenstrukturen fest, die bei einer Änderung des Parallelitätsgrades umverteilt werden müssen.

set(SOURCE, int source_id, typ *source_addr) – Definiert die Adresse des ersten Elementes der Datenstruktur, die umverteilt werden muß. In einer einzelnen Einstellungsumgebung entspricht die Anzahl der *Source*-Spezifizierungen der Anzahl der umzuverteilenden Datenstrukturen, d.h. der o.g. *NumSource*.

set(VIRT, int virt_degree) – Entspricht dem initialen Virtualisierungsgrad der einzigen Datenstrukturen. Auch hier gibt es in einer Einstellungsumgebung dem Parameter *NumSource* entsprechend viele *Virt*-Spezifizierungen.

set(NUMPAR, int num) – Legt die Anzahl der Programm-Variablen fest, deren aktuellen Werte der *Main Worker* an die neuaktivierten *Worker* bei einer Erhöhung des Parallelitätsgrades mitteilen muß. Ein typisches Beispiel dafür ist die aktuelle Iteration der parallelen Schleife.

set(PAR, int par_id, typ *par) – Definiert die Adresse der Programm-Variablen, deren Werte bei einer Erhöhung des Parallelitätsgrades an die neuaktivierten *Worker* mitgeteilt werden müßen.

set(PAR_VALUE, int par_id, int var_value) – Spezifiziert den Wert der Schleifenvariable für die letzte Iteration der parallelen Schleifen. Dadurch erkennt das System ParGrad, daß die Einstellung des Parallelitätsgrades der entsprechenden Einstellungsumgebung zu Ende ist.

set(NUMPARLOCAL, int local_num) – Legt die Anzahl der lokalen Parameter fest, die bei einer Änderung des Parallelitätsgrades aktualisiert werden müssen.

set(PARLOCAL, int par_id, typ *par_addr) – Definiert die Adresse der Parameter, die bei einer Änderung des Parallelitätsgrades lokal aktualisiert werden müssen. In einer einzelnen Einstellungsumgebung gibt es der Anzahl *NumParLocal* entsprechend viele *ParLocal*-Spezifizierungen.

set(DATA_DISTRIBUTION, int distribution_type) – Spezifiziert die Art der Datenverteilung, d.h. ob die Elemente blockweise oder zyklisch auf die Prozessoren verteilt sind bzw. umverteilt werden.

set(DATA_STRUCTURE, int data_type) – Spezifiziert die Art der Datenstrukturen, die bei einer Änderung des Parallelitätsgrades umverteilt werden müssen. Die durch ParGrad unterstützten Strukturen sind z.B. Felder und Matrizen (siehe Abscnhitt 5.7.3).

set(PERIOD, int value) – Legt die Häufigkeit der Einstellung des Parallelitätsgrades fest, d.h. den Abstand der Einstellungen, wobei der Parameter in *Iterationen der parallelen Schleife* gemessen wird. Wenn dem Parameter *Frequency* z.B. 1 zugewiesen wird, passiert die Einstellung in jeder Iteration der Schleife. Hier gibt es zwei konträre Aspekte: Auf der einen Seite ist es wünschenswert, daß die Kosten der Einstellung gering bleiben. Daraus folgt, daß der Parallelitätsgrad nicht so oft geändert werden soll, d.h. dem Parameter *Period* wird ein hoher Wert zugewiesen. Auf der anderen Seite ist es wichtig, daß der optimale Parallelitätsgrad so schnell wie möglich gefunden wird, was nur mit einem möglichst geringen Wert für den Parameter *Period* erreicht werden kann. Der Benutzer soll bei der Spezifizierung des *Frequency*-Parameters einen Kompromiß zwischen diesen beiden konträren Aspekten finden.

set(STRATEGY, int strategy_id) – Spezifiziert die Strategie, wonach der Parallelitätsgrad eingestellt wird. Die ausgewählte Strategie legt das Ziel der Einstellung fest (siehe Abschnitt 5.5.3).

set(EFFTHRESHOLD, float effi) – Definiert die Effizienzschranke, deren Unterschreitung von den Strategien EISS, SLEVS und EGSS (siehe Abschnitt 5.5) als Kriterium für die Einstellung betrachtet wird.

set(SPDTHRESHOLD, float spd) – Definiert die Beschleunigungsschranke, deren Unterschreitung von der Strategie SGSS (siehe Abschnitt 5.5) als Kriterium für die Einstellung betrachtet wird.

set(FILENAME, char *file_name) – Spezifiziert den Namen der Datei, für die die Einstellung des Parallelitätsgrades vorgenommen wird. Dieser Name wird auch für die entsprechenden Protokoll-Dateien (siehe Abschnitt 6.11) und für die Parallelitätsgrad-Datenbank (siehe Abschnitt 5.9) benutzt.

Für eine Inter-Lauf-Einstellung (siehe Abschnitt 5.8) sind noch die zwei folgenden Parameter zu spezifizieren:

set(NUMCALLS, int num_methods) – Definiert die Anzahl der Methoden, die bei einer Inter-lauf-Einstellung aufzurufen sind.

set(CALLS, int call_id, typ *method) – Definiert die Adressen der Methoden, die bei einer Inter-lauf-Einstellung aufzurufen sind. Die Anzahl der Adressen ist dem Parameter *NumCalls* gleich.

Falls der Benutzer nichts spezifiziert, nimmt das System ParGrad an, daß es sich um eine Intra-Lauf-Einstellung mit blockweiser Datenverteilung von Feldern unter der Strategie RISS mit Häufigkeit 1 handelt.

6.7.3 Änderungen im Programm

Neben der Instrumentierung des parallelen Programmes müssen u.U. bis zu drei Änderungen im Programm vorgenommen werden, um seine Ausführung mit der Einstellung des Parallelitätsgrades durchführen zu können. Diese drei Änderung werden im folgenden beschrieben.

Falls die vom Programm benutzte Prozessoranzahl mittels der *shmem*-Routine *_num_pes()* im Benutzerprogramm innerhalb einer Einstellungsumgebung abgefragt wird, muß diese Abfrage durch einen Aufruf auf die ParGrad-Routine *get(DEGREE)* ersetzt werden. Das ist notwendig, weil sich der Parallelitätsgrad innerhalb der Einstellungsumgebung ändert und daher von der maximalen Prozessoranzahl (Wert, der von der Routine *_num_pes()* geliefert wird) abweichen kann.

Die zweite Änderung bezieht sich auf die Aktualisierung des Virtualisierungsgrades. Wenn es im Benutzerprogramm innerhalb einer Einstellungsumgebung vorkommt, daß der Virtualisierungsgrad einer Datenstruktur abgefragt wird, muß die Abfrage durch einen Aufruf der ParGrad-Routine *get(VIRT)* ersetzt werden. Diese Routine liefert den aktuellen Wert für den Virtualisierungsgrad.

Die dritte Änderung hat mit der Ausführung einer Synchronisation zu tun, die innerhalb einer Einstellungsumgebung im Benutzerprogramm plaziert ist und an der alle vom

Benutzer allozierten Prozessoren teilnehmen. In diesem Fall muß die *shmem*-Routine *shmem_barrier_all* durch die Routine *shmem_barrier* ersetzt werden, die eine eingeschränkte Anzahl der an der Synchronisation teilnehmenden Prozessoren erlaubt. Das ist bei der Einstellung des Parallelitätsgrades notwendig, weil die aktuelle Anzahl der benutzten Prozessoren häufig von der maximalen Anzahl abweicht.

Ein Beispiel wird im folgenden gezeigt, wobei in der Zeile 2 der Zugriff auf die Variable *Keys_per_Proc* durch den Aufruf *get(VIRT)* ersetzt wurde. In der Zeile 8 wurde der Aufruf auf die *shmem*-Routine *shmem_barrier_all* durch den Aufruf auf die Routine *shmem_barrier* ersetzt. Zudem wurde in der Zeile 8 der Aufruf *get(DEGREE)* anstatt *_num_pes* benutzt.

```
1      . . .
2      for (i = 0 ; i < get(VIRT) ; i++)
3      {
4       buckets[((keys[i] << shift) >> shift2)]++;
5       offset[((keys[i] << shift) >> shift2)]++;
6      }
7      . . .
8      shmem_barrier(MASTER, 0, get(DEGREE), pSync_barrier);
```

6.8 Verwaltung der Einstellung des Parallelitätsgrades

Hier werden das Prinzip und die Algorithmen präsentiert, die bei der Verwaltung der Einstellung des Parallelitätsgrades eingesetzt wurden. Der *SystemModule* (siehe Abbildung 6.2) ist für die Verwaltung zuständig.

6.8.1 Grundidee

ParGrad-Prozesse können aktiv oder inaktiv sein, je nachdem, was der aktuelle Parallelitätsgrad beträgt. Am Anfang der Ausführung eines Programmes unter der Einstellung des Parallelitätsgrades sind alle durch den Benutzer allozierten PEs bzw. ParGrad-Prozesse aktiv. Bei einer Einstellung kann der Parallelitätsgrad erniedrigt werden, womit einige aktive Prozesse inaktiv werden. Bei einer Erhöhung des Parallelitätsgrades können inaktive ParGrad-Prozesse durch den Prozeß_0 (*Main Worker*) wieder aktiviert werden, wie in Abbildung 6.4 gezeigt. Im System sind immer $[Parallelitätsgrad - 1]$ ParGrad-Prozesse aktiv.

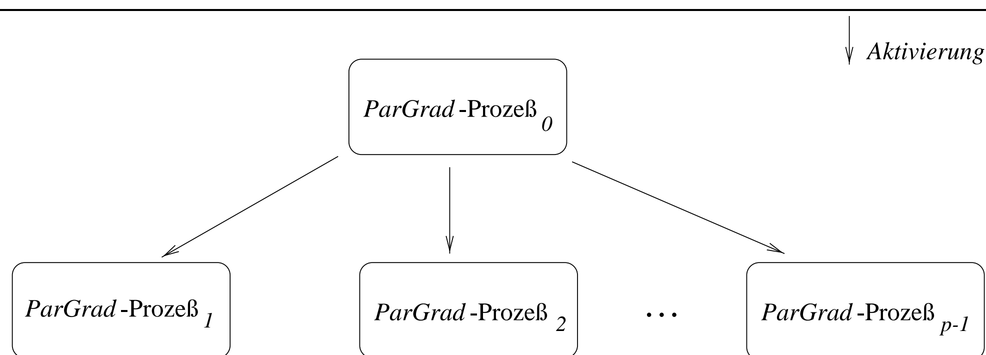


Abbildung 6.4: ParGrad-Prozesse

Zu jedem ParGrad-Prozeß gehört eine Liste von Einstellungsumgebungen, wie in Abbildung 6.5 gezeigt. Diese Liste hat mindestens ein Element, wobei zu jedem Zeitpunkt nur genau eine Einstellungsumgebung pro ParGrad-Prozeß aktiv ist.

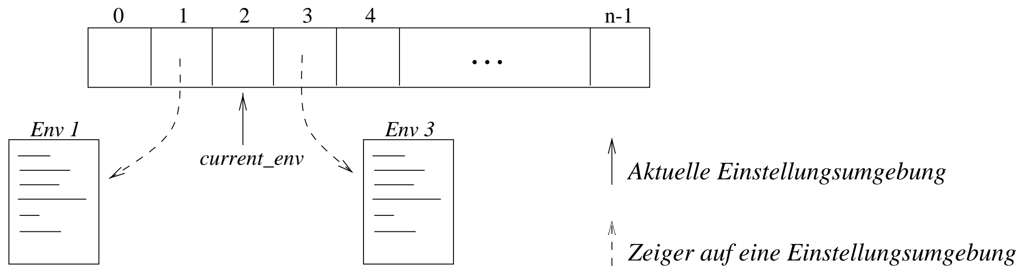


Abbildung 6.5: Liste der Einstellungsumgebungen bei einem ParGrad-Prozeß

Es werden so viele *Environment*-Objekte (siehe Abschnitt 6.5) erzeugt wie es Einstellungsumgebungen im Klassenkonstruktor des Benutzerprogrammes gibt (siehe Beispiel im Abschnitt 6.7.2). Die aktive Einstellungsumgebung wird durch den Zeiger *current_ env* festgehalten. Die Bezeichner der Umgebungen im Benutzerprogramm entsprechen dem jeweiligen Index der Einstellungsumgebung in der Liste. Jedesmal, wenn die Einstellungsumgebung gewechselt wird, wird *current_ env* aktualisiert.

6.8.2 Intra-Lauf-Einstellung

Hier werden die Algorithmen präsentiert, die die Verwaltung der Intra-Lauf-Einstellung (siehe Abschnitt 5.8) des Parallelitätsgrades implementieren. Der erste Algorithmus (siehe Abbildung 6.6) wird durch den *Main Worker* ausgeführt.

```
void MainWorker::ParGrad()
{
     time_stop = clock();
     if (par_degree not in data base)
          if (iteration % Frequency)
               time_with_degree = time_stop - time_start;
               degreeadapter->UpdateDegreePar();
               if (*par_degree != old_degree)
                    distributer->NewDistribution();
               end if
          end if
     end if
     if (CheckTerminationLoop())
          WriteOutDataBase(par_degree);
          AbortParGrad();
          RestoreStateBeforeAdap();
     end if
     time_start = clock();
}
```

Abbildung 6.6: Algorithmus: Verwaltung der Intra-Lauf-Einstellung beim *Main Worker*

Zuerst wird die Uhr gestoppt, damit genau die für die Ausführung der parallelen Schleife

im Benutzerprogramm benötigte Zeit aufgenommen wird.[7] Falls sich der Parallelitätsgrad der betroffenen Schleife nicht in der Parallelitätsgrad-Datenbank (siehe Abschnitt 5.9) befindet und die jetzige Iteration der parallelen Schleife mit Einstellung duchgeführt werden soll (abhängig von *Period*-Parameter), findet der Einstellungsschritt mittels eines Aufrufes der Methode *UpdateDegreePar* der Klasse *DegreeAdapter* statt. Falls der Parallelitätsgrad geändert wird, wird die Datenumverteilung mittels eines Aufrufes der Methode *NewDistribution* der Klasse *Distributer* ausgelöst. Danach wird überprüft, ob die jetzige Iteration der parallelen Schleife die letzte ist. In diesem Fall wird der Zustand, der vor dem Beginn der Einstellung des Parallelitätsgrades aktuell war, wiederhergestellt. Dieser initiale Zustand entspricht einer Datenverteilung über alle zum Anfang allozierten Prozessoren.

Der zweite Algorithmus (siehe Abbildung 6.7) wird durch die normalen *Worker* ausgeführt.

```
void Worker::ParGrad()
{
    do
        while ((mesg = distributer->ReadMesg()) == NO_DATA)
        switch (mesg)
            . . .
            case NEW_DATA: new_degree = *par_degree;
            . . .
        end switch
    while ((my_pe() >= *par_degree) && (mesg != KILL));          /* inactive PE waiting */
    if ((mesg == KILL) || (CheckTerminationLoop())
        RestoreStateBeforeAdap();
    end if /* else Worker is active now */
}
```

Abbildung 6.7: Algorithmus: Verwaltung der Intra-Lauf-Einstellung bei einem *Worker*

Ein *Worker* bleibt solange inaktiv, d.h. nimmt an der Ausführung der parallelen Schleife nicht teil, bis der Parallelitätsgrad größer wird als sein Prozessor-Bezeichner (wird durch einen Aufruf auf die Routine *my_pe()* ermittelt) oder eine *Kill*-Nachricht vom *Main Worker* eingetroffen ist. Weil Prozessoren nicht flexibel aufgefordert bzw. freigegeben werden können, wird der inaktive Zustand durch den Leerlauf der inaktiven *Worker* (durch die *While*-Anweisung in Abbildung 6.7) simuliert. Ein inaktiver *Worker* wird vom *Main Worker* durch eine *New_Data*-Nachricht aktiviert. Der Zustand, der vor der Einstellung des Parallelitätsgrades aktuell war, wird ggf. wiederhergestellt.

6.8.3 Inter-Lauf-Einstellung

Hier werden die Algorithmen vorgestellt, die die Verwaltung der Inter-Lauf-Einstellung (siehe Abschnitt 5.8) des Parallelitätsgrades implementieren. Der erste Algorithmus (siehe Abbildung 6.8) wird durch den *Main Worker* ausgeführt.

Falls der Parallelitätsgrad des Programmes nicht in der Parallelitätsgrad-Datenbank (siehe Abschnitt 5.9) im Rahmen einer früheren Einstellung gespeichert wurde, wird das komplette Programm mehrmals ausgeführt, wobei der Parallelitätsgrad zwischen den Läufen

[7]Dies ist die erste Operation in der ParGrad-Methode, weil der Aufruf auf ParGrad am Ende der parallelen Schleife im Benutzerprogramm plaziert ist. Am Ende des Einstellungsschrittes wird die Uhr wieder gestartet, so daß für die konkrete Einstellung (durch die Methode *UpdateDegreePar* realisiert) nur die genaue Ausführungszeit der parallelen Schleife berücksichtigt wird.

```
void MainWorker::ParGrad()
{
    if (par_degree not in data base)
        for all adaptation iterations
            multicasts(NEW_DATA, current_iteration) to PE_1 until PE_{new_degree-1};
            time_start = clock();
            calls program;
            time_stop = clock();
            time_with_degree = time_stop - time_start;
            degreeadapter->UpdateDegreePar();
        end for
        WriteOutDataBase(par_degree);
    end if
    AbortParGrad();
}
```

Abbildung 6.8: Algorithmus: Verwaltung der Inter-Lauf-Einstellung beim *Main Worker*

eingestellt wird. Die Anzahl der Läufe wird durch den Benutzer über einen Parameter bei der Instrumentierung spezifiziert. Bei jedem Lauf des Programmes sendet der *Main Worker* den aktuellen Iterationsbezeichner und eine *New_ Data*-Nachricht an jeden aktiven *Worker*, d.h. an die *Worker*, die an PE_1 bis $PE_{Parallelitätsgrad-1}$ plaziert sind. Die Dauer einer jeder Ausführung des Programmes wird aufgenommen. Dieser Zeitdauer wird bei der konkreten Einstellung des Parallelitätsgrades, die mittels eines Aufrufes der Methode *UpdateDegree-Par* der Klasse *DegreeAdapter* erfolgt, von der Einstellungsstrategie benutzt. Am Ende der Einstellungsiterationen wird der gefundene optimale Parallelitätsgrad mittels der Methode *WriteOutDataBase* in die Datenbank gespeichert und die Einstellung durch die Methode *AbortParGrad* beendet.

Der zweite Algorithmus wird durch die normalen *Worker* ausgeführt (siehe Abbildung 6.9).

```
void Worker::ParGrad()
{
    do
        while ((mesg = distributer->ReadMesg()) == NO_DATA);
        if (mesg != KILL)
            calls program;
        end if
    while ((current_iteration < total_iterations) && (mesg != KILL));
}
```

Abbildung 6.9: Algorithmus: Verwaltung der Inter-Lauf-Einstellung beim *Worker*

Ein *Worker* bleibt solange inaktiv, d.h. nimmt an der Ausführung der parallelen Schleife nicht teil, bis er eine Nachricht vom *Main Worker* bekommt. Wenn es sich dabei um keine *Kill*-Nachricht handelt, fängt ein neuer Lauf des Programmess an. Die Einstellung ist zu Ende, wenn die durch den Benutzer spezifizierte Anzahl der Läufe des Programmes erreicht wurde oder wenn eine *Kill*-Nachricht, die vom *Main Worker* gesendet wurde, empfangen wird.

Der *Main Worker* bzw. die *Worker* wissen durch den Parameter *Iterations* (siehe Abschnitt 6.7.2), ob es sich um eine Intra-Lauf- oder eine Inter-Lauf-Einstellung handelt. Wenn

der Benutzer diesen Parameter durch den Klassenkonstruktor initialisiert hat, nimmt das System ParGrad an, daß es sich um eine Inter-Lauf-Einstellung handelt.

6.9 Einstellung des Parallelitätsgrades

Hier werden das Prinzip und die Algorithmen diskutiert, die bei der Realisierung der konkreten Einstellung des Parallelitätsgrades eingesetzt wurden. Der *DegreeAdapter* (siehe Abbildung 6.2) ist für die konkrete Einstellung zuständig.

6.9.1 Grundidee

Wie in den Algorithmen in Abbildungen 6.6 bzw. 6.8 gezeigt wurde, wird die konkrete Einstellung des Parallelitätsgrades vom *Main Worker* ausgelöst, wobei der *DegreeAdapter* für ihre Ausführung zuständig ist. Die Auslösung dieser Operation wird mittels eines Aufrufes auf die *UpdateDegreePar*-Methode der Klasse *DegreeAdapter* realisiert. Der neue Parallelitätsgrad wird nach einer vom Benutzer ausgewählten Einstellungsstrategie berechnet und dem *Main Worker* zurückgegeben. Auf die Realisierung dieser Strategien wird im folgenden näher eingegangen.

6.9.2 Algorithmen

Hier werden Algorithmen für vier Strategien zur Einstellung des Parallelitätsgrades vorgestellt. Als Eingabe bekommen die Strategien den aktuellen Parallelitätsgrad, die Ausführungszeit der parallelen Schleife mit einem Prozessor und die Ausführungszeit der parallelen Schleife mit dem aktuellen Parallelitätsgrad, wobei die Strategien RISS und PeSS die Information über die Ausführungszeit der parallelen Schleife mit einem Prozessor nicht brauchen (siehe Abschnitt 5.5). Als Ausgabe haben alle Strategien den neu-berechneten Parallelitätsgrad. Die in den folgenden Algorithmen benutzten Variablen *minproc* bzw. *maxproc* enthalten die minimale bzw. maximale Prozessoranzahl, wobei die minimale Prozessoranzahl 1 ist und die maximale beim Starten des Programmes durch den Benutzer spezifiziert wird.

Die **E***fficiency-driven* **I***ncremental* **S***earch* **S***trategy* (EISS) ist in Abbildung 6.10 vereinfacht dargestellt. Der Parallelitätsgrad wird so lange halbiert, bis sich die Effizienz oberhalb der vorgegebenen Effizienzschranke befindet. Wenn dieses Ziel erreicht wurde, wird der Parallelitätsgrad für die nächsten Iterationen der parallelen Schleife nicht mehr geändert.

Die Grundzüge der **R***untime* **I***ncremental* **S***earch* **S***trategy* (RISS) sind in Abbildung 6.11 dargestellt. Der Parallelitätsgrad wird so lange halbiert, bis die Ausführungszeit der parallelen Schleife dadurch nicht mehr niedriger wird, sondern wieder zu wachsen anfängt. Wenn dieser Punkt erreicht ist, wird der Parallelitätsgrad für die nachfolgenden Iterationen der parallelen Schleife nicht mehr geändert.

Die **C***ost*-**B***enefit-driven* **I***ncremental* **S***earch* **S***trategy* (CBISS) ist in Abbildung 6.12 vereinfacht aufgeführt. Der Parallelitätsgrad wird so lange halbiert, bis die Nutzen-Kosten-Relation durch die Reduzierung des Parallelitätsgrades nicht mehr höher wird. Sobald dieser Punkt erreicht wird, bleibt der Parallelitätsgrad für die nachfolgenden Iterationen der parallelen Schleife konstant.

Abbildung 6.13 zeigt das Grundschema der **P***rediction-driven* **S***earch* **S***trategy* (PeSS). Der Parallelitätsgrad wird so lange halbiert, bis an einem Punkt die für die nächste Halbierung vorhergesagte Ausführungszeit der parallelen Schleife wieder zu wachsen anfängt.

```
int DegreeAdapter::EISS()
{
     if (OptimalDegreeParallelism not found yet)
          .  .  .
          speed_up = time_with_1 / time_with_degree;
          efficiency = speed_up / par_degree;
          if (efficiency < current_env->EffThreshold)
               par_degree = max(minproc,par_degree/2);
          else
               OptimalDegreeParallelism = found;
          end if
     end if
     return(par_degree);
}
```

Abbildung 6.10: Algorithmus: Einstellung des Parallelitätsgrades unter der EISS-Strategie

```
int DegreeAdapter::RISS()
{
     if (OptimalDegreeParallelism not found yet)
          .  .  .
          if (time_with_degree <= old_runtime)
               par_degree = max(minproc,par_degree/2);
          else
               par_degree = min(maxproc,par_degree*2);
               OptimalDegreeParallelism = found;
          end if
     end if
     return(par_degree);
}
```

Abbildung 6.11: Algorithmus: Einstellung des Parallelitätsgrades unter der RISS-Strategie

Ab disem Punkt wird der Parallelitätsgrad für die nachfolgenden Iterationen der parallelen Schleife nicht mehr geändert. Falls die Anzahl der möglichen Punkte bzw. Parallelitätsgrade die minimale Anzahl 4 unterschreitet, wird die Strategie RISS anstatt der Strategie PeSS eingesetzt.

6.10 Datenumverteilung

Hier werden das Prinzip und die Algorithmen präsentiert, die bei der Realisierung der Datenumverteilung eingesetzt wurden. Der *Distributer* (siehe Abbildung 6.2) ist für die Datenumverteilung zuständig.

6.10.1 Grundidee

Wie die Abbildung 6.14 zeigt, sind alle aktiven und alle durch eine Erhöhung des Parallelitätsgrades aktivwerdenden ParGrad-Prozesse an einer Datenumverteilung beteiligt. Wenn der Parallelitätsgrad durch den *DegreeAdapter* bei einer Intra-Lauf-Einstellung geändert wird, löst der *Main Worker* (*ParGrad-Prozeß*$_0$) durch einen Aufruf der *NewDistribution*-Methode (siehe Abbildung 6.6) der Klasse *Distributer* eine Datenumverteilung aus.

```
int DegreeAdapter::CBISS()
{
    if (OptimalDegreeParallelism not found yet)
        . . .
        speed_up = time_with_1 / time_with_degree;
        efficiency = speed_up / par_degree;
        benefit_cost = time_with_1 / (pow(time_with_degree,2) * par_degree);
        if (benefit_cost > old_bc)
            par_degree = max(minproc,par_degree/2);
        else
            par_degree = min(maxproc,par_degree*2);
            OptimalDegreeParallelism = found;
        end if
    end if
    return(par_degree);
}
```

Abbildung 6.12: Algorithmus: Einstellung des Parallelitätsgrades nach der CBISS-Strategie

Der *Distributer* am *Main Worker* sendet eine Nachricht (Export- oder Import-Befehl) an jeden *Worker*, um ihnen individuell mitzuteilen, ob sie bei der Datenumverteilung Daten exportieren oder importieren werden. Dies wird durch die gestrichelten Linien in Abbildung 6.14 dargestellt und in Abbildung 6.15 als Algorithmus repräsentiert. In diesem Algorithmus überprüft der *Main Worker* mit dem Zweck, die exportierenden *Worker* zu definieren bzw. eine Export-Nachricht an sie zu schicken, ob es sich bei der aktuellen Einstellungsumgebung um eine blockweise oder zyklische Datenverteilung handelt. Nachher führt er selber eine Export- oder Import-Operation aus, je nachdem, ob der Parallelitätsgrad erhöht oder erniedrigt wurde. Als letztes werden die importierenden *Worker* definiert und ihnen eine Import-Nachricht gesendet.

Aufgrund der empfangenen Nachricht löst jeder *Worker* die entsprechenden Operation aus, nämlich eine Export- oder eine Import-Operation, wie in dem Algorithmus in Abbildung 6.16 erläutert und anhand der gepunkteten Linien in Abbildung 6.14 dargestellt wird. Wenn ein *Worker* eine Export-Nachricht empfängt, löst er eine Datenexportierung aus und überprüft, ob er trotz der Änderung des Parallelitätsgrad noch aktiv bleiben wird. Falls dies der Fall ist, löst er eine Datenimportierung aus, wenn der Parallelitätsgrad niedrig geworden ist (d.h. der Virtualisierungsgrad ist größer geworden, jeder aktive *Worker* wird mehr Daten importieren als exportieren).

Die Export- bzw. Import-Operationen werden durch den *Distributer* eines jeden *Worker* bzw. des *Main Worker* ausgeführt, welches anhand der durchgezogenen Linien in Abbildung 6.14 gezeigt wird. Die Datenumverteilung wird im folgenden Abschnitt näher erläutert.

6.10.2 Datenaustausch zwischen *Worker*

Bei der Spezifizierung einer Einstellungsumgebung wird durch den Benutzer dem Parameter *distribution_mode* ein Wert zugewiesen. Anhand des *distribution_mode*-Parameters wird festgelegt, aus welcher Subklasse der Klasse *Distributer* (siehe Abbildung 6.3) das Objekt, das für die Datenumverteilung zuständig ist, erzeugt wird. Insgesamt hat die Klasse *Distributer* vier vorgegebene Subklassen, nämlich *Vector*, *Matrix*, *Cube* und *Struct* (siehe Abschnitt 6.5). Zusätzlich kann der Benutzer eine weitere Subklasse mit Import- bzw. Export-Methoden

```
int DegreeAdapter::PeSS()
{
    if (OptimalDegreeParallelism not found yet)
        if (GetLog2(maxproc) <= min_points)   /* Minimal number of points on the curve = 4 */
            par_degree = max(minproc,par_degree/2);
            current_env->strategy = RISS;
        else
            AddCurvePoints(par_degree,time_with_degree);
            if (current_num_points >= min_points)
                FillMatrix();
                Gauss();
                pred_time = TimePrediction(max(minproc,par_degree/2));
                if (pred_time < time_with_degree)
                    par_degree = max(minproc,par_degree/2);
                else
                    OptimalDegreeParallelism = found;
                end if
            else
                par_degree = max(minproc,par_degree/2);
            end if
        end if
    end if
    return(par_degree);
}
```

Abbildung 6.13: Algorithmus: Einstellung des Parallelitätsgrades unter der PeSS-Strategie

spezifizieren, falls eine vom ParGrad-System nicht unterstützte Datenstruktur umverteilt werden soll. Im folgenden werden die Algorithmen für eine Daten-Exportierung (siehe Abbildung 6.17) bzw. -Importierung (siehe Abbildung 6.19) präsentiert, wobei ein Feld die jeweils umzuverteilende Datenstruktur darstellt. Das System ParGrad nimmt immer ein Feld als Datenstruktur an, wenn der Benutzer keine andere Datenstruktur explizit angibt. Die Export- bzw. Import-Operationen für die anderen vom ParGrad-System unterstützten Datenstrukturen sind analog zu den hier für Felder präsentierten Operationen.

6.10.2.1 Export-Operation

Hier wird der Algorithmus für die Daten-Exportierung (siehe Abbildung 6.17) erläutert.

Bei einer Daten-Exportierung wird zuerst der Virtualisierungsgrad aller umzuverteilenden Datenstrukturen aktualisiert, je nachdem ob die Exportierung aufgrund einer Erhöhung oder Erniedrigung des Parallelitätsgrades ausgelöst wurde. Falls es sich um eine blockweise Datenverteilung handelt, wird danach die Anzahl der Elemente pro Block festgelegt, bevor die Indizes der Elemente konvertiert werden und Daten gesendet werden.

Die Daten werden unter der Benutzung der Cray-*shmem*-Bibliothek (siehe Abschnitt 6.3) gesendet, wobei die Routinen *shmem_put* und *shmem_iput* eingesetzt werden. Die Routine *shmem_put* wird zur Übertragung der Daten an einen entfernten Prozessor bei einer blockweisen Verteilung benutzt. Der Aufruf der Routine erfolgt durch die Spezifizierung von vier Parametern, nämlich der Adresse des Blockanfangs am entfernten Prozessor, der Adresse des Blockanfangs am lokalen Prozessor, der Anzahl der zu übertragenden Elemente und dem Bezeichner des entfernten Prozessors. Auf der anderen Seite wird die Routine *shmem_iput* für die Übertragung der Daten mit Übersprung (*strided data*) an einen entfernten Prozessor bei einer zyklischen Verteilung verwendet. Der Aufruf dieser Routine erfolgt durch die

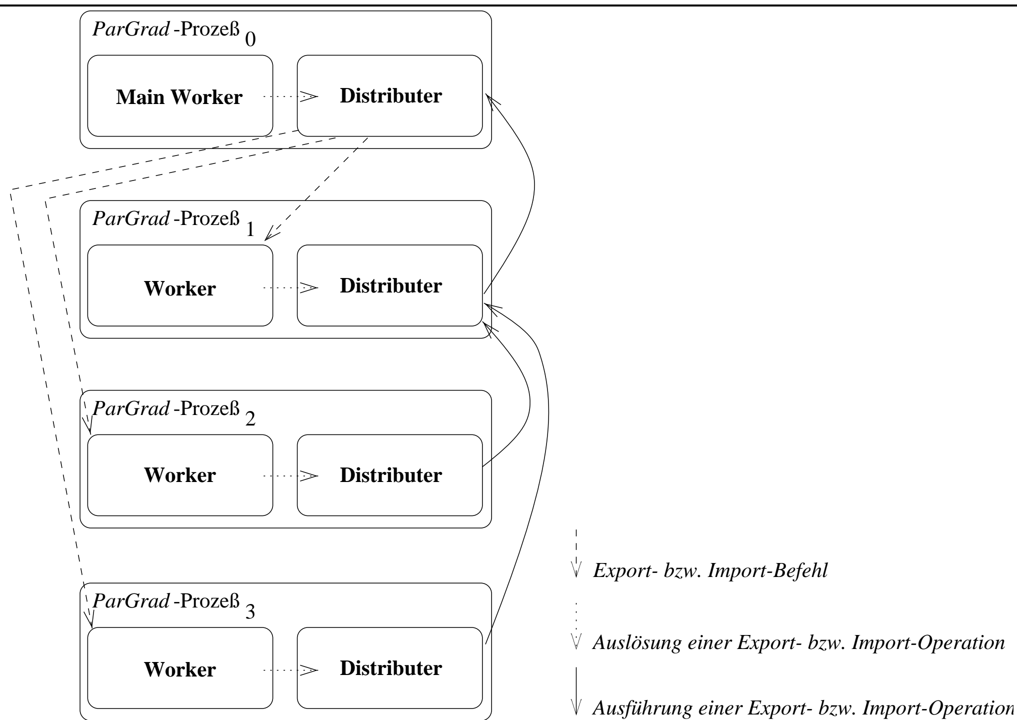


Abbildung 6.14: Datenumverteilung über die ParGrad-Prozesse. Dies entspricht der Datenumverteilung, die bei dem Beispiel der Abbildung 5.6 in Kapitel 5 vorgenommen werden muß.

Spezifizierung von sechs Parametern, nämlich der Adresse des Blockanfangs am entfernten Prozessor, der Adresse des Blockanfangs am lokalen Prozessor, der Schrittweite (*stride*) am entfernten Prozessor, dergleichen am lokalen Prozessor, der Anzahl der zu übertragenden Elemente und dem Bezeichner des entfernten Prozessors.

Falls es sich in der aktuellen Einstellungsumgebung um eine zyklische Datenverteilung handelt und die Änderung des Parallelitätsgrades einer Erhöhung desselben bzw. einer Erniedrigung des Virtualisierungsgrades entspricht, muß der Exporteur, nachdem er die Daten an den Importeur geschickt hat, eine Komprimieroperation durchführen. Diese Operation hat die Abschaffung des durch die Daten-Exportierung entstandenen Speicherplatzes zwischen Elementen als Ziel (siehe Abbildung 6.18).

6.10.2.2 Import-Operation

Hier wird der Algorithmus für die Daten-Importierung vorgestellt (siehe Abbildung 6.19).

Falls es sich in der aktuellen Einstellungsumgebung um eine zyklische Datenverteilung handelt und die Änderung des Parallelitätsgrades einer Erniedrigung desselben bzw. einer Erhöhung des Virtualisierungsgrades entspricht, muß der Importeur eine Schiebeoperation (*shift*) durchführen, bevor er die Daten vom Exporteur empfangen kann. Diese Operation hat die Freistellung vom Speicherplatz für die zu importierenden Daten als Ziel (siehe Abbildung 6.20).

Nachdem der Importeur die Daten bekommen hat, muß er noch den Virtualisierungsgrad aller importierten Datenstrukturen aktualisieren.

```
void Distributer::NewDistribution()
{
      . . .
                                                                /* Message to the exporters... */
      if (current_env->data_distribution == BLOCK)
            start_pe = 1;
      else                                                          /* cyclic data distribution */
            if (new_degree > old_degree)
                  start_pe = 1;
            else
                  start_pe = new_degree;
            end if
      end if
      for PE = start_pe until old_degree - 1
            SendMesg(PE,EXPORT);
      end for
                                            /* Main Worker is also an exporter or an importer */
      if (new_degree > old_degree)
            Export_Data();
      else
            Import_Data();
      end if
                                                                /* Message to the importers... */
      if (current_env->data_distribution == BLOCK)
            start_pe = old_degree;
      else                                                          /* cyclic data distribution */
            if (new_degree > old_degree)
                  start_pe = old_degree;
            else
                  start_pe = 1;
            end if
      end if
      for PE = start_pe until new_degree - 1
            SendMesg(PE,IMPORT);
      end for
      . . .
}
```

Abbildung 6.15: Algorithmus: Signalisierung einer Import- bzw. Export-Operation an jeden *Worker* bzw. an den *Main Worker*

6.11 Ausgabe der Einstellung

Das ParGrad-System sieht zwei Ausgaben-Arten zur Darstellung der Ergebnisse vor, die aus der Ausführung eines parallelen Programmes mit der Einstellung des Parallelitätsgrades resultieren. Die erste Art entspricht mehreren Protokoll-Dateien, die jeweils Daten über verschiedene Einstellungsumgebungen enthalten und die Erweiterung *.his* haben. Von jeder einzelnen Einstellungsumgebung werden die partielle Effizienz, die partielle Ausführungszeit und der partielle Mehraufwand (oder Teileffizienz, Teilausführungszeit und Teilmehraufwand) der Einstellung gespeichert, wobei sich diese *partiellen* Werte jeweils auf die Ausführung einer Gruppe von Iterationen der parallelen Schleife mit demselben Parallelitätsgrad beziehen. So sind die Werte dieser Kenngröße für jeden eingesetzten Parallelitätsgrad bekannt und liefern ein Profil der Gesamtausführung mit der Einstellung des Parallelitätsgrades. Die in Kapitel 7 präsentierten Ergebnisse basieren auf dem Daten in der Protokoll-Datei.

```
void Worker::ParGrad()
{
    do
        while ((mesg = distributer->ReadMesg()) == NO_DATA)
        switch (mesg)
            . . .
            case EXPORT:
                distributer->Export_Data();
                if (my_pe() < new_degree)
                    distributer->Import_Data();
                    if (new_degree < old_degree)
                        distributer->Import_Data();
                    end if
                end if
            case IMPORT:
                distributer->Import_Data();
            . . .
        end switch
    while ((my_pe() >= new_degree) && (mesg != KILL));            /* PE keeps inactive */
    . . .
}
```

Abbildung 6.16: Algorithmus: Auslösen einer Import- bzw. Export-Operation durch einen *Worker*

Diese Ausgabe-Art wird im folgenden anhand des Beispiels in Abbildung 6.1 näher erläutert.

Tabelle 6.1: Beispiel einer Protokoll-Datei

Parallelitätsgrad	1	2	4
Efficiency	1.00	0.44	0.17
Runtime	226168.00	128338.00	166075.00
Overhead	1300.00	274.00	542.00

Das Beispiel zeigt das Ergebnis der Ausführung eines parallelen Programmes mit der Einstellung des Parallelitätsgrades, wobei das Programm mit vier Prozessoren gestartet wurde. Zu sehen sind jeweils drei Werte für die Effizienz, die Ausführungszeit und den Mehraufwand, der durch die Benutzung des ParGrad-Systems entsteht. In jeder Zeile bezieht sich der erste Wert auf die Iterationen der parallelen Schleife, die mit einem Prozessor ausgeführt wurden, der zweite Wert auf Iterationen mit zwei Prozessoren, und der dritte Wert mit vier Prozessoren, wobei die Ausführungszeit und die ParGrad-Mehraufwand in Millisekunden angegeben sind.

Die zweite Ausgabe-Art der Ergebnisse sieht eine Datei vor, die Teil der Parallelitätsgrad-Datenbank ist (siehe Abschnitt 5.9) und die Erweiterung *.dat* hat. Jeder Eintrag in der Datei enthält Daten über eine vollständige Einstellung des Parallelitätsgrades in einer Einstellungsumgebung. Ein Beispiel einer Datenbank-Datei ist in Abbildung 6.2 zu sehen.

Das Beispiel zeigt das Ergebnis für drei Ausführungen eines parallelen Programmes mit der Einstellung des Parallelitätsgrades, wobei das Programm in allen präsentierten Fällen mit 64 Prozessoren gestartet wurde. In jeder Zeile sind sechs Werte enthalten, die sich auf

```
void Distributer::Export_Data()
{
for all data structures which will be redistributed
      UpdateVirtualizationDegree();
end for
if (current_env->data_distribution == BLOCK)
      if (new_degree > old_degree)
            how_many_elements = current_env->VirtualizationDegree;
      else                                                  /* new_degree < old_degree! */
            how_many_elements = old_VirtualizationDegree;
      end if
end if
.   .   .
for all data structures which will be redistributed
                /* the next for-instruction ist only necessary for a data_distribution = blockwise */
      for all blocks of elements to be exported step how_many_elements
            global_index = local2global(element_index);
            pe = index2PE(global_index);
            if (pe != my_pe())
                  start_remoteindex = global2local(global_index);
                  if (current_env->data_distribution == BLOCK)
                        if (new_degree > old_degree)
                              shmem_put(&data_structure[start_remoteindex],
                              &data_structure[block_begin],current_env->VirtualizationDegree,pe);
                        else                                  /* new_degree < old_degree */
                              shmem_put(&data_structure[start_remoteindex],
                              &data_structure[block_begin],old_VirtualizationDegree,pe);
                        end if
                  else                                        /* data_distribution = cyclic */
                        if (new_degree > old_degree)
                              shmem_iput(&data_structure[start_remoteindex],&data_structure[1],
                              1,new_degree/old_degree,current_env->VirtualizationDegree,pe);
                              Compresses the remaining data in the local PE;
                        else                                  /* new_degree < old_degree */
                              shmem_iput(&data_structure[start_remoteindex],&data_structure[0],
                              old_degree/new_degree,1,old_VirtualizationDegree,pe);
                        end if
                  end if
            end if
      end for
end for
}
```

Abbildung 6.17: Algorithmus: Export-Operation, die jeder Exporteur ausführt

die folgenden Aspekte beziehen:

- Datum der Programmdatei
- die maximale Anzahl von Prozessoren bei der betrachteten Ausführung
- der Bezeichner für die parallele Schleife, für die der Parallelitätsgrad eingestellt wurde
- die eingesetzte Einstellungsstrategie
- die Problemgröße

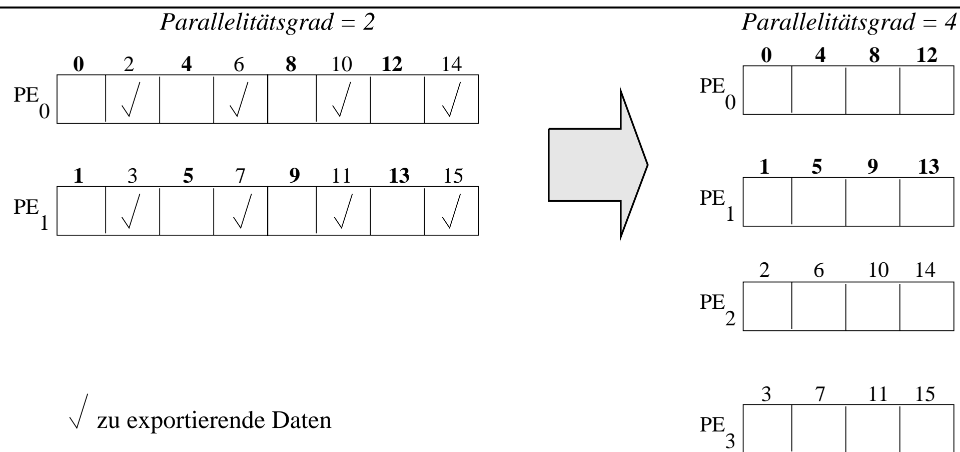


Abbildung 6.18: Beispiel einer Komprimierung bei zyklischer Datenverteilung; Erhöhung des Parallelitätsgrades

```
void Distributer::Import_Data()
{
      . . .
      if ((current_env->data_distribution == CYCLIC) && (new_degree < old_degree))
            Shifts data in the local PE before receiving data;
            /* if the parallelism degree has become smaller and the virtualization degree
            therefore higher when using cyclic data distribution */
      end if
      for all data structures
            UpdateVirtualizationDegree();
      end for
}
```

Abbildung 6.19: Algorithmus: Import-Operation, die durch jeden Importeur ausgeführt wird

- der gefundene optimale Parallelitästgrad

Der Parallelitätsgrad der drei verschiedenen Ausführungen des Programmes wurde nach unterschiedlichen Strategien eingestellt, nämlich RISS, EISS bzw. PeSS (unter 9, 1 bzw. 8 in der vierten Spalte in Abbildung 6.2 zu lesen).

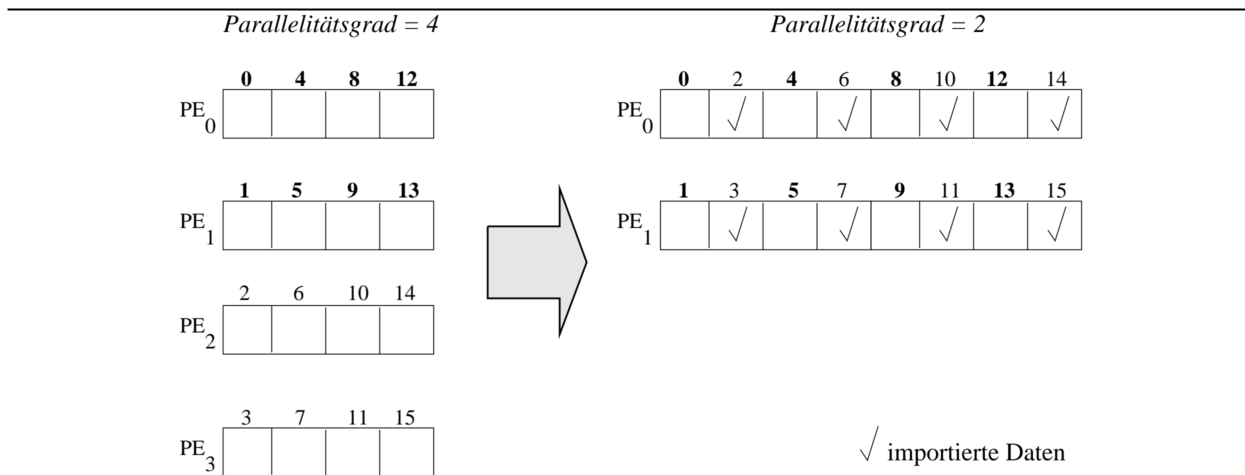


Abbildung 6.20: Beispiel einer Schiebeoperation bei einer zyklischen Datenverteilung; Erniedrigung des Parallelitätsgrades

Tabelle 6.2: Beispiel einer Datenbank-Datei

Date	Max. PE	Parallel Loop	Strategy	Problemsize	Optimal Degree
7/7/1999, 18:52	64	1	9	65536	4
7/7/1999, 18:55	64	1	1	65536	4
7/7/1999, 19:01	64	1	8	65536	4

Kapitel 7

Ergebnisse

Hier werden Ergebnisse gezeigt, die durch die Einstellung des Parallelitätsgrades von Programmen erzielt wurden, und durch die die Thesen dieser Arbeit (siehe Abschnitt 1.2) validiert wurden. Ergebnisse für die zwei im Rahmen dieser Arbeit entwickelten Methoden zur Bestimmung des optimalen Parallelitätsgrades werden vorgestellt, nämlich die Methode, die den optimalen Parallelitätsgrad über das mathematische Modell bestimmt (siehe Abschnitt 4) und die, die das Suchverfahren dafür einsetzt (siehe Kapitel 5). Zuerst wird die Benchmark-Sammlung, auf die die Ergebnisse basieren, präsentiert. Danach erfolgt für beide Arten der Bestimmung des Parallelitätsgrades jeweils die Vorstellung der eingesetzten Validierungstechnik sowie eine detaillierte Diskussion der erzielten Ergebnisse. Zudem werden die Ergebnisse der mathematischen Modell den über Suchverfahren erzielten Ergebnisse gegenüber gestellt.

7.1 Benchmark-Sammlung

Die Benchmark-Sammlung besteht aus Programmen, die die Bereiche Mathematik, Geophysik und Elektrotechnik vertreten. Die Benchmark-Sammlung setzt sich aus einem Sortieralgorithmus, drei Kerne der *Livermore Loops* sowie aus drei kompletten Anwendungen der o.g. Bereiche zusammen. Die Programme der Benchmark-Sammlung werden im folgenden präsentiert, wobei jeweils der Anwendungsbereich, der Algorithmus, die Problemgröße, die Parallelitätsquelle, das Kommunikationsmuster sowie die Größe und Quelle des Programmes vorgestellt werden.

Die Benchmark-Sammlung enthält Vertreter für jede in Abschnitt 4.2 präsentierte Programmklasse. Dies ermöglicht eine vollständige Analyse der Einstellung des Parallelitätsgrades im Bezug auf die Programmklassen. Die drei ausgewählten *Livermore Loops* sind die vorher noch fehlenden Vertreter der Klassen K1 und K4 sowie ein weiterer Vertreter der Klasse K2, bei dem als Besonderheit der Kommunikationsaufwand linear mit dem Verhältnis zwischen der Problemgröße und der Anzahl der Prozessoren wächst. Die restlichen *Livermore Loops* wurden nicht auf die Einstellung des Parallelitätsgrades untersucht, weil dies lediglich zu einer Wiederholung der bereits berücksichtigten Fälle geführt hätte (siehe Anhang A).

7.1.1 Radix-Sortieralgorithmus

Anwendung: Parallele Sortierung ist eine wichtige Grundoperation für viele Anwendungen im Bereich Parallelverarbeitung [48], z.B. Anwendungen aus der Mathematik, Graphik

und Simulation. Sortierung ist eine der am meisten untersuchten Probleme in der Informatik [57].

Algorithmus: Der Radix-Sortieralgorithmus [57, 55, 48] ist eine iterative Sortiermethode, bei der eine Iteration pro Ziffer des Schlüssels durchgeführt wird, wie am Beispiel in Abbildung 7.1 gezeigt wird. In dem Beispiel werden 8 Zahlen bestehend jeweils aus drei Ziffern in drei Iterationen von rechts nach links (*straight-radix sort* [38]) zifferweise sortiert.

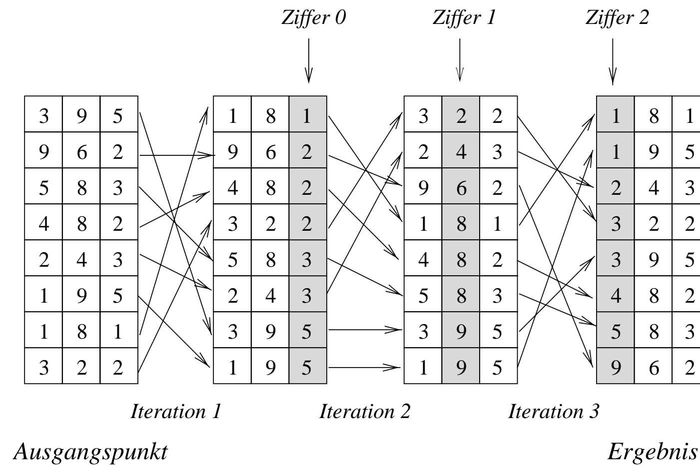


Abbildung 7.1: Beispiel einer Radix-Sortierung von Zahlen, die aus drei Ziffern bestehen

Bei einer parallelen Ausführung des Algorithmus werden Schlüssel unter den Prozessoren verteilt. In jeder Iteration analysiert ein Prozessor die ihm zugeordneten Schlüssel. Dabei generiert er ein lokales Histogramm, das für eine bestimmte Ziffer der ihm zugeordneten Schlüssel festhält, wie oft jeder möglicher Wert darin vorkommt. Wenn die Schlüssel in Abbildung 7.1 z.B. blockweise auf zwei Prozessoren verteilt werden (PE_0 erhält die obersten 4 Schlüssel in Abbildung 7.1), sieht das Histogramm bzgl. der *Ziffer 0* für jeden Prozessor nach der ersten Iteration (*Iteration 1*) wie in Abbildung 7.2 aus.

	0	1	2	3	4	5	6	7	8	9
PE_0	0	1	3	0	0	0	0	0	0	0

	0	1	2	3	4	5	6	7	8	9
PE_1	0	0	0	2	0	2	0	0	0	0

Abbildung 7.2: Lokale Histogramme für das Beispiel einer Radix-Sortierung

Die lokalen Histogramme werden zu einem globalen zusammengesetzt, indem eine Summe des Histogrammes stellenweise über alle Prozessoren gebildet wird. Dieses globale Histogramm wird von jedem Prozessor benutzt, um lokale Schlüssel auszutauschen bzw. eine neue Schlüsselstruktur für die nächste Iteration zu generieren.

Problemgröße: Hier hängt die Problemgröße von der Anzahl der zu sortierenden Schlüssel, von der Anzahl der Ziffern, aus denen jeder Schlüssel besteht, und von der Anzahl der unterschiedlichen Werte ab, die eine Ziffer übernehmen kann (*radix*).

Quelle des Parallelismus: Da jeder Prozessor ein lokales Histogramm besitzt, kann jede Iteration parallel ausgeführt werden, wobei eine Iteration den für eine Ziffer benötigten Berechnungen entspricht. Zwischen der Generierung des lokalen und des globalen Histogrammes, sowie zwischen der Generierung des globalen Histogrammes und dem Austausch von Schlüsseln unter den Prozessoren ist eine Synchronisation der Prozessoren notwendig.

Kommunikationsmuster: Die Generierung des globalen Histogrammes benötigt eine Reduktionsoperation für die Summe über alle Prozessoren. Außerdem entspricht das Vertauschen der Schlüssel einem *all-to-all*-Kommunikationsmuster.[1]

Größe: ca. 530 LOC

Quelle: Die Implementierung dieser Anwendung wurde aus einem Benchmark von Split-C [11] abgeleitet. Die Aufrufe auf Split-C-Primitiven wurden durch Aufrufe von Cray-Routinen ersetzt. Eine Postfix-Summe wurde entwickelt, um den Index der lokalen Schlüssel in einer Iteration zu bestimmen. Das Programm wurde in C++ umgeschrieben. Für die Inter-Prozessor-Kommunikation wurde die Cray-*shmem*-Bibliothek benutzt.

7.1.2 *Livermore Loop 1* (LL1)

Anwendung: *LL1* ist eins der Kerne, das zu den *Livermore Loops* [17] gehört. Es handelt sich um ein Fragment eines Programmes aus dem Bereich Hydrodynamik.

Algorithmus: Der Wert für x_k wird anhand der folgenden Formel berechnet:

$$x_k = Q + [Y_k \times [R \times Z_{k+10} + T \times Z_{k+11}]]$$

wobei $k = 1, ..., n$ und n der Anzahl der zu berechnenden Elemente entspricht.

Problemgröße: Die Problemgröße wird hier durch die Anzahl der Elemente k gegeben.

Quelle des Parallelismus: Da kein x-Element in der Berechnung von x_k vorkommt, gibt es keine Datenabhängigkeit, d.h. die Berechnung eines jeden Elementes von x kann parallel durchgeführt werden.

Kommunikationsmuster: Jeder Prozessor muss sich maximal 11 Elemente vom Nachbarprozessor holen, um die Berechnung fortzusetzen.

Größe: ca. 170 LOC

Quelle: Die Implementierung dieses Programmes wurde aus der Arbeit von Matthias Müller [40] abgeleitet.

[1]Bei der *all-to-all*-Kommunikationsmuster kommuniziert jeder Prozessor mit allen anderen Prozessoren.

7.1.3 *Livermore Loop 3* (LL3)

Anwendung: *LL3* aus den *Livermore Loops* [17] berechnet das Skalarprodukt (*inner product*) zweier Vektoren. Diese Operation tritt in wissenschaftlichen Problemen sehr häufig auf.

Algorithmus: Die partiellen Summen werden lokal berechnet; danach werden die partiellen Ergebnisse durch eine Reduktion über alle Prozessoren aufsummiert.

Problemgröße: Die Problemgröße ist durch die Länge der Vektoren gegeben.

Quelle des Parallelismus: Alle partiellen Summen können parallel berechnet werden.

Kommunikationsmuster: Eine Summe-Reduktion über alle Prozessoren wird durchgeführt.

Größe: ca. 140 LOC

Quelle: Die Implementierung dieses Programmes wurde aus der Arbeit von Matthias Müller [40] abgeleitet.

7.1.4 *Livermore Loop 13* (LL13)

Anwendung: *LL13* aus den *Livermore Loops* [17] ist ein Fragment eines *2-D Particle-in-Cell*-Codes [17].

Algorithmus: *LL13* besteht aus einer mehrfachen Aktualisierung des Vektors H, wobei dieser Vektor über alle Prozessoren verteilt ist. Der Zugriff auf die Vektor-Elemente geschieht über eine indirekte Indizierung.

Problemgröße: Die Problemgröße ist hier durch die Länge der Vektoren gegeben.

Quelle des Parallelismus: Alle außer der letzten Iteration der Schleife können parallel ausgeführt werden.

Kommunikationsmuster: Die Aktualisierung des Feldes H setzt das Holen von Elementen verschiedener Felder voraus, wobei die Felder blockweise über die Prozessoren verteilt sind. Das Kommunikationsmuster ist irregulär, weil es erst zur Laufzeit feststeht, welche Elemente geholt werden sollen.

Größe: ca. 280 LOC

Quelle: Die Implementierung dieses Programmes wurde aus der Arbeit von Matthias Müller [40] abgeleitet.

7.1.5 Veltran-Operator

Anwendung: Der Veltran-Operator ist ein geophysikalischer Operator für die Analyse spezifischer Geschwindigkeiten von Schallwellen in unterschiedlichen Materialien der inneren Erdschichten. Er wird gewöhnlich als erstes Glied einer Kette von Operatoren eingesetzt, an deren Ende ein Profil der Erdschichten als Ergebnis produziert wird. Das Ziel dabei ist beispielsweise die Erkundung von Erdöl- und Erdgasvorkommen in den inneren Erdschichten.

Algorithmus: Der Veltran-Operator [30, 31] verarbeitet rohe Meßdaten, die von seismographischen Empfängern aufgezeichnet werden, wie in Abbildung 7.3 dargestellt. Die Schallwellen entstehen am Punkt S, wobei *x0* bis *x7* unabhängige seismographische Empfänger sind. Die gestrichelten Linien bezeichnen die Grenzen zwischen zwei horizontalen Erdschichten *l1* und *l2*, die aus unterschiedlichen Materialien bestehen bzw. in denen sich eine Schallwelle mit einer jeweils unterschiedlichen spezifischen Geschwindigkeit ausbreitet. An Meßpunkt x_i werden die Signalstärken aufgezeichnet und dadurch Quellmatrizen gebildet.

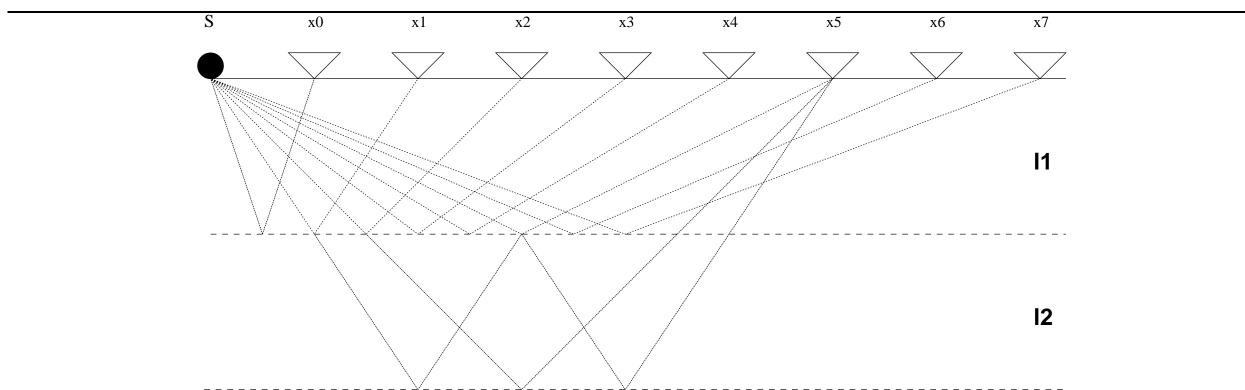


Abbildung 7.3: Messungen seismischer Daten

Zur Berechnung der Geschwindigkeiten wird die Zeitspanne benötigt, die der Dauer für die Signalausbreitung vom Punkt S senkrecht nach unten bis zu einer Schichtgrenze und wieder zurück entspricht. Diese Zeitspanne kann aber nicht gemessen werden, da es nicht möglich ist, einen Empfänger direkt an den Punkt S zu plazieren. Folge daraus ist, daß Veltran die spezifischen Geschwindigkeiten und die Tiefe der Schichten aus den Meßdaten ableiten muß. Die Berechnung wird somit folgendermaßen durchgeführt: Veltran iteriert potentielle Geschwindigkeiten aus einem Bereich von n_v Werten. Für jede dieser Geschwindigkeiten werden hypothetische Hyperbeln berechnet, danach werden die Signalmessungen aus der Quellmatrix entlang dieser Hyperbeln aufsummiert. Für diejenige hypothetische Hyperbel, die mit der Hyperbel innerhalb der Quellmatrix übereinstimmt, erreicht die Summe ein Maximum und nähert daher die gesuchte Geschwindigkeit am besten an. Die Berechnung der Summe benötigt n_t (Anzahl der Zeilen) Schritte, da jeder Zeitpunkt während des Experiments betrachtet werden muß. Daher benötigt Veltran für eine einzelne Quellmatrix $n_v.n_x.n_t$ Schritte, wobei n_x der Anzahl der Spalten entspricht.

Zur Konstruktion eines zweidimensionalen Bildes des Erdinneren werden Messungen für mehrere Erzeugungspunkte durchgeführt, die sich auf einer Linie an der Erdoberfläche befinden. Daraus resultiert eine Menge von Quellmatrizen, die zu einem dreidimensionalen Würfel angeordnet werden.

Problemgröße: Hier entspricht die Problemgröße der Anzahl der Elemente der dreidimensionalen Matrix.

Quelle des Parallelismus: Die Eingabedaten sind in einer dreidimensionalen Matrix gespeichert und werden für die parallele Ausführung über die y-Achse zwischen den Prozessoren verteilt. Jeder Prozessor ist für die Berechnung der ihm zugeordneten Spalten zuständig.

Kommunikationsmuster: Für die lokale Berechnung braucht jeder Prozessor auch Daten, die sich in anderen Spalten bzw. anderen Prozessoren befinden.

Größe: ca. 770 LOC

Quelle: Die Implementierung dieser Anwendung wurde aus der Fortran-Version der Arbeit von Matthias Jacob [30] abgeleitet. Die Anwendung wurde in C++ umgeschrieben, wobei die Cray-*shmem*-Bibliothek für die Kommunikation eingesetzt wurde.

7.1.6 TLM-Verfahren

Anwendung: Das TLM(*Transmission Line Matrix*)-Verfahren ist eine Methode zur numerischen Feldberechnung, die im Bereich Elektrotechnik für die Analyse der elektromagnetischen Verträglichkeit zwischen elektronischen Geräten eingesetzt wird.

Algorithmus: Um über den Algorithmus des TLM-Verfahrens [18, 22] diskutieren zu können, müssen zuerst einige Datenstrukturen eingeführt werden. Ein *Knoten* dient zur Diskretisierung eines Raumgebiets, wobei er die Eigenschaften des Feldgebiets innerhalb eines Elementes modelliert. Die *Anregung* beschreibt, welcher Knoten mit welcher Anregungsfunktion angeregt werden soll. Der *Empfänger* gibt die Orte für die Ausgabe wieder, d.h. an welchen Orten die Wellenamplituden ausgelesen und verarbeitet werden. Der *Randknoten* enthält Angaben darüber, an welchem Ort ein Knoten an eine Materialgrenze stößt und welcher Transmissions- bzw. Reflexionsfaktor dort existiert. Der Algorithmus betreffend die Aktivitäten in einem Zeitschritt ist im folgenden vorgestellt, wobei die Anzahl der Iterationen parametrisiert wird:

1 Schreibe eine Anregung in die Knoten
2 Führe eine Randknotenstreuung aus
3 Führe Wellenfortpflanzung aus
4 Führe Knotenstreuung aus
5 Gebe Empfänger aus
6 Kehre zum Schritt 1 zurück

Zum Schritt 1: An dieser Stelle wird in die Ports bestimmter Knoten eine Wellenamplitude geschrieben.

Zum Schritt 2: Falls ein Knoten an eine Materialgrenze stößt, steht der Reflexionsfaktor bzw. der Transmissionsfaktor zur Verfügung, damit die für die Ports einfallenden Werte berechnet werden können.

Zum Schritt 3: Hier wird erkannt, daß ein einfallender Impuls eines Knotens im nächsten Zeitschritt zu einem einfallenden Impuls des angrenzenden Knotens wird. Nach einem Austausch mit dem Nachbarknoten liegen nur noch einfallende Impulse an den Ports der Knoten.

Zum Schritt 4: Wenn nur noch einfallende Impulse an den Ports eines Knotens liegen, kann die eigentliche Berechnung der Streumatrix stattfinden. Nach diesem Schritt liegen nur noch ausfallende Werte an den Ports.

Zum Schritt 5: Nach der Knotenstreuung liegen die Wellenamplituden eines Zeitpunktes vor und die Werte können aus den Empfangspunkten ausgelesen werden.

Problemgröße: Die Problemgröße wird hier durch die Anzahl der Knoten und der Iterationen gegeben.

Quelle des Parallelismus: Die Berechnung der Streumatrix für die Randknotenstreuung ist die Funktion im TLM-Verfahren, die parallelisiert werden kann. Die Streumatrix wird für jeden Knoten und jeden Zeitschritt neu berechnet. Jeder Prozessor bekommt einen Teil des gesamten Raumes, wie in Abbildung 7.4 dargestellt.

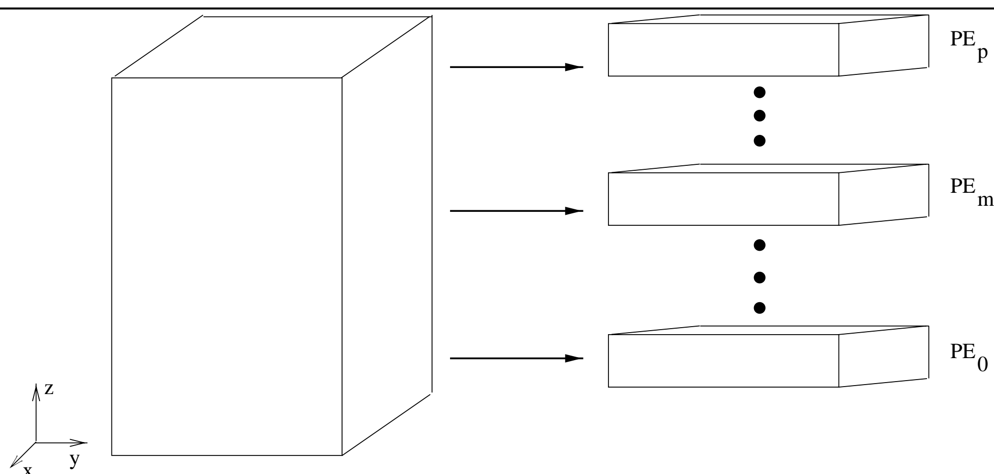


Abbildung 7.4: Verteilung eines Feldproblemes auf p Prozessoren

Kommunikationsmuster: Zwecks der Wellenfortpflanzung müssen die Prozessoren miteinander kommunizieren. Das Muster der Inter-Prozessor-Kommunikation für diese Anwendung entspricht einer Nachbar-Nachbar-Kommunikation.

Größe: ca. 1370 LOC

Quelle: Die Implementierung dieser Anwendung wurde aus der Arbeit von Peter Fischer und Martin Gebhardt [22, 18] abgeleitet. Die Anwendung wurde in C++ umgeschrieben. Für die Kommunikation zwischen Prozessoren wurde die Cray-*shmem*-Bibliothek benutzt.

7.1.7 PDE-Löser

Anwendung: Auf Gebieten wie der Strömungsmechanik (Turbulenzforschung), der Phasenübergänge (Kristallwachstum), der Diffusions-Reaktion (Flammenausbreitung) sowie der Wetterprognose leisten partielle Differentialgleichungen wichtige Beiträge zum Verständnis von Naturphänomenen. Darüber hinaus finden mathematische Modelle dieser Art auch immer stärkeren Eingang in der Medizin und Sozialwissenschaften.

Algorithmus: Der hier betrachtete PDE (*Partial Differential Equation*)-Löser setzt dreidimensionale *Fast Fourier Transformation* (3D-FFT) ein, um partielle Differentialgleichungen zu lösen. Für die Beschreibung des Algorithmus wird hier angenommen, daß die Eingabe eine Matrix A der Form n_1 x n_2 x n_3 ist. Die Berechnung besteht aus vier Phasen, die im folgenden erläutert werden:

1. Berechnung einer 1D-FFT an n_3 Punkten für alle n_1 x n_2 komplexen Vektoren.

2. Berechnung einer 1D-FFT an n_2 Punkten für alle n_1 x n_3 komplexen Vektoren.

3. Austausch des Erbegnis-Vektors in einen n_2 x n_3 x n_1 komplexen Vektor B.

4. Ausführung einer 1D-FFT an n_1 Punkten für alle n_2 x n_3 komplexen Vektoren.

Problemgröße: Die Problemgröße wird hier durch die Anzahl der Elemente der dreidimensionalen Matrix und die Anzahl der Iterationen des Algorithmus definiert.

Quelle des Parallelismus: Die Berechnung aller Phasen kann parallel durchgeführt werden, wobei die Prozessoren erst in der Austausch-Phase kommunizieren müssen. Die Daten werden über die erste Dimension der Eingabe-Matrix A über die Prozessoren verteilt, d.h. jeder Prozessor bekommt n_1/p Zeilen der Matrix A zu berechnen, wobei p der Gesamtanzahl der Prozessoren entspricht. Wenn n_1/p gleich i ist, bekommt jeder Prozessor die Elemente $A_{i,j,k}$, wobei $0 \leq j < n_2$ und $0 \leq k < n_3$ gilt.

Kommunikationsmuster: Die Prozessoren müssen in der Austausch-Phase (*tranposition phase*) miteinander kommunizieren. Das Kommunikationsmuster entspricht einer *all-to-all*-Kommunikation.

Größe: ca. 1500 LOC

Quelle: Die Implementierung dieser Anwendung basiert auf [37] und auf der *NAS benchmark suite* [2, 3]. Die Anwendung wurde in C++ umgeschrieben, wobei die Cray-*shmem*-Bibliothek für die Inter-Prozessor-Kommunikation eingesetzt wurde.

7.2 Einstellung über das mathematische Modell

Die Ergebnisse der Bestimmung des Parallelitätsgrades über das mathematische Modell wurden durch den Einsatz der Methode erzielt, die in Kapitel 4 vorgestellt wurde. Im folgenden wird die Validierungstechnik definiert, die bei der Zusammenstellung der Ergebnisse zugrunde lag. Danach erfolgt die Präsentation der Benchmark-Sammlung anhand der Programmklassen und eine Diskussion der Ergebnisse. Dieser Abschnitt beschäftigt sich außerdem mit der Analyse der Genauigkeit der durch das mathematische Modell gelieferten Ergebnisse sowie mit der Extrapolation aus den Ergebnissen.

7.2.1 Validierungstechnik

Mehrere Programme mit unterschiedlichen Kommunikationsprofilen stellen eine Benchmark-Sammlung zusammen, die als Basis bei der Untersuchung der Einstellung des Parallelitätsgrades über das mathematische Modell diente. Der optimale Parallelitätsgrad wurde anhand der Ausführungszeit, der Effizienz sowie der Nutzen-Kosten-Relation berechnet, wobei die Funktionen für den Kommunikations- bzw. Berechnungsaufwand eines jeden Programmes und die Latenzzeit auf der Cray-T3E (1483 ns) bekannt waren[2]. Die Untersuchung fand mit der in Abschnitt 4.1 präsentierten Methode, unter der Berücksichtigung mehrerer unterschiedlicher Problemgrößen für jedes Programm der Benchmark-Sammlung, statt. Der über das mathematische Modell berechnete optimale Parallelitätsgrad wurde mit dem realen Parallelitätsgrad verglichen. Die Bestimmung des realen Parallelitätsgrades eines jeden in der Benchmark-Sammlung enthaltenen Programmes erfolgte durch Messungen der Ausführungszeit auf der Cray-T3E. Die Ausführungszeit eines jeden Programmes wurde für jede berücksichtigte Problemgröße mit einer variierenden Anzahl von Prozessoren auf der Cray gemessen, so daß die Parallelitätsgrade, die zu der minimalen Ausführungszeit, der vorgegebenen Effizienz sowie der maximalen Nutzen-Kosten-Relation geführt haben, bestimmen werden konnten.

Wie in Kapitel 4 ausführlich erläutert wurde, ist das Ziel der Einstellung des Parallelitätsgrades über das mathematische Modell, den Parallelitätsgrad so einzustellen, daß das Minimum der Ausführungszeit, die vorgegebene Effizienz oder das Maximum der Nutzen-Kosten-Relation erreicht wird.

7.2.2 Programmklassen der Benchmark-Sammlung

Die Benchmark-Sammlung besteht aus sieben Programmen, die die Bereiche Mathematik, Geophysik und Elektrotechnik vertreten. Es handelt sich um drei Anwendungen der o.g. Bereiche, einen Sortieralgorithmus sowie drei *Livermore Loops* [17]. Die Benchmark-Sammlung wird im folgenden zusätzlich zu der detaillierten Präsentation in Abschnitt 7.1 anhand der Programmklassen bzgl. des Kommunikationsaufwandes diskutiert.

LL1: Dieses Programm (siehe Abschnitt 7.1.2) gehört zur Klasse K1 (siehe Abschnitt 4.2), d.h. es hat eine konstante Kommunikationszeit. Für einen Virtualisierungsgrad größer als 9 werden immer 11 nicht-lokale Speicherzugriffe durchgeführt.

Radix-Sortieralgorithmus: Dieses Programm (siehe Abschnitt 7.1.1) gehört zur Klasse K2 (siehe Abschnitt 4.2), d.h. es hat eine Kommunikationszeit, die linear mit der Anzahl der Prozessoren sowie mit der Problemgröße wächst.

LL13: Dieses Programm (siehe Abschnitt 7.1.4) gehört zur Klasse K2 (siehe Abschnitt 4.2), d.h. seine Kommunikationszeit wächst linear mit dem Verhältnis zwischen der Problemgröße und der Anzahl der Prozessoren.

TLM: Dieses Programm (siehe Abschnitt 7.1.6) vertritt die Klasse K2 (siehe Abschnitt 4.2), d.h. die Kommunikationszeit wächst linear mit der Problemgröße.

[2]Die Latenzzeit auf der Cray-T3E wurde den Messungen in [40] entnommen.

PDE-Löser: Dieses Programm (siehe Abschnitt 7.1.7) gehört zur Klasse K3 (siehe Abschnitt 4.2), d.h. die Kommunikationszeit wächst quadratisch mit der Anzahl der Prozessoren.

LL3: Dieses Programm (siehe Abschnitt 7.1.3) gehört zur Klasse K4 (siehe Abschnitt 4.2), d.h. seine Kommunikationszeit wächst logaritmisch mit der Anzahl der Prozessoren.

Veltran-Operator: Der Kommunikationsaufwand dieses Programmes (siehe Abschnitt 7.1.5) wird einer Kombination der Klassen K2 und K4 (siehe Abschnitt 4.2) zugeordnet. Er wächst linear mit dem Verhältnis zwischen der Problemgröße und der Anzahl der Prozessoren sowie logarithmisch mit der Anzahl der Prozessoren.

7.2.3 Diskussion der Ergebnisse

Hier werden die Ergebnisse der Einstellung des Parallelitätsgrades über das mathematische Modell (siehe Kapitel 4) anhand der Benchmark-Sammlung diskutiert. Zur Erinnerung ist anzumerken, daß die Ausführungszeit eines parallelen Programmes durch die Formel

$$T_p = \frac{T_1}{p} + T_{comm} \tag{7.1}$$

die Effizienz durch die Formel

$$E_p = \frac{T_1}{p \times T_p} \tag{7.2}$$

und die Nutzen-Kosten-Relation durch die Formel

$$R_{BC} = \frac{T_1}{T_p^2 \times p} \tag{7.3}$$

gegeben sind (siehe Abschnitt 4.1), wobei T_1 die Ausführungszeit mit einem Prozessor und T_p die Ausführungszeit mit p Prozessoren darstellt.

Die Ergebnisse der Einstellungen des Parallelitätgrades werden in Tabellenform vorgestellt. In der ersten Spalte einer Tabelle ist die Problemgröße aufgezeichnet, die zweite Spalte gibt den über das mathematische Modell berechneten optimalen Parallelitätsgrad wieder. Die dritte Spalte stellt den realen optimalen Parallelitätsgrad dar, der durch Messungen bestimmt wurde. Die letzte Spalte zeigt den Fehler, der durch die Auswirkung einer falschen Einstellung des Parallelitätsgrades verursacht wird. Dieser Fehler bezieht sich je nach Ziel der Einstellung auf die Ausführungszeit, Effizienz oder Nutzen-Kosten-Relation des parallelen Programmes. Die falsche Einstellung beruht auf der Abweichung zwischen dem über das mathematische Modell berechneten und dem realen optimalen Parallelitätsgrades für jede berücksichtigte Problemgröße.

7.2.3.1 LL1

Der Kommunikationsaufwand für *LL1* ist konstant, wenn der Virtualisierungsgrad größer als 11 ist (siehe Abschnitt 7.1.2). Die Ausführungszeit mit p Prozessoren (T_p) ist für *LL1* somit durch die folgende Formel gegeben:

$$T_p = \frac{T_1}{p} + (11 \times niter \times M) \tag{7.4}$$

wobei *niter* die Anzahl der Iterationen des Algorithmus und M die Latenzzeit pro entfernten Zugriff (Kommunikationsoperation) darstellen. Die Effizienz wird anhand der Formel 7.2 und 7.4 bestimmt:

$$E_p = \frac{T_1}{T_1 + 11 \times niter \times M \times p} \geq Effizienzschranke$$

In diesem Fall gilt $p \leq \frac{T1 \times (1-E)}{11 \times E \times niter \times M}$, wobei E für die Effizienzschranke steht. Basierend auf den Formeln (7.3) und (7.4) wird die Nutzen-Kosten-Relation wie folgt berechnet:

$$R_{BC} = \frac{T_1 \times p}{(T_1 + 11 \times p \times niter \times M)^2}$$

Die Nullstelle der ersten Ableitung für R_{BC} bzw. das Maximum der Nutzen-Kosten-Relation wird beim folgenden Parallelitätsgrad erreicht: $p = \frac{T_1}{11 \times niter \times M}$.

Tabelle 7.1: Ergebnisse der Einstellung des Parallelitätsgrades für $LL1$, wenn $niter = 100$

Livermore Loop 1			
Problemgröße	Optimaler Parallelitätsgrad		Fehler
	Berechneter Wert	Realer Wert	
Ziel: Minimierung der Ausführungszeit			
2 K	256	128	23,0%
4 K	256	256	0,0%
32 K	4096	512[a]	31,0%
Ziel: Einhaltung einer Effizienzschranke von 50%			
2 K	15	16	0,1%
4 K	31	32	0,1%
32 K	260	128	10,0%
Ziel: Maximierung der Nutzen-Kosten-Relation			
2 K	16	32	5,1%
4 K	32	64	1,0%
32 K	257	160	18,8%

[a] Der reale Wert konnte für diesen Fall nicht gemessen werden, da es nur bis zu 512 Prozessoren zur Verfügung standen.

Tabelle 7.1 zeigt die Ergebnisse, die für die Einstellung des Parallelitätsgrades über das mathematische Modell für *LL1* erzielt wurden. Für weniger umfangreiche Problemgrößen

wie 2K oder 4K Elemente bleibt die Auswirkung der Abweichung zwischen dem berechneten und dem realen optimalen Parallelitätsgrad auf die Ziele der Einstellung gering, nämlich im Durchschnitt kleiner als 5%. Bei einer umfangreicheren Problemgröße wie 32K Elementen steigt der Fehler bzgl. der Kennzahlen bis auf ca. 19%. Dies ist vermutlich auf Cache-Effekte zurückzuführen, da der Bedarf an Speicherplatz für solche Problemgrößen sehr hoch wird (LL1 braucht 3 Vektoren, jeder bestehend aus 32K Elementen, wobei jedes Element 8 Byte benötigt). Cache-Effekte werden in den Berechnungs- bzw. Kommunikationsaufwänden nicht berücksichtigt, welches die Genauigkeit des über das mathematische Modell berechneten Parallelitätsgrades verschlechtert.

7.2.3.2 Radix-Sortieralgorithmus

Der Kommunikationsaufwand für den Radix-Sortieralgorithmus wächst linear mit der Anzahl der Prozessoren sowie mit der Anzahl der zu sortierenden Schlüssel. Die Ausführungszeit mit p Prozessoren (T_p) ist:

$$T_p = \frac{T_1}{p} + z \times \left(p \times radix + N + \frac{N}{p} \right) \times M \tag{7.5}$$

wobei z der Anzahl der Ziffern pro Schlüssel, $radix$ der Anzahl der unterschiedlichen Werte, die eine Ziffer übernehmen kann, N der Anzahl der Schlüssel und M der Latenzzeit pro entfernten Zugriff entsprechen. Somit erreicht die Ausführungszeit ihr Minimum bei $p = \frac{\sqrt{z \times M \times radix \times (T_1 + z \times M \times N)}}{z \times M \times radix}$. Die Effizienz wird anhand der Formel 7.2 und 7.5 bestimmt:

$$E_p = \frac{T_1}{T_1 + z \times M \times p^2 \times radix + z \times M \times N \times p + z \times M \times N} \geq Effizienzschranke$$

Basierend auf den Formeln (7.3) und (7.5) wird die Nutzen-Kosten-Relation wie folgt berechnet:

$$R_{BC} = \frac{T_1 \times p}{(T_1 + z \times p^2 \times radix \times M + z \times N \times p \times M + z \times N \times M)^2}$$

Das Maximum der Nutzen-Kosten-Relation wird beim folgenden Parallelitätsgrad erreicht: $p = \frac{-N \times z \times M \pm \sqrt{N^2 \times M^2 \times z^2 + 12 \times z \times T_1 \times radix \times M + 12 \times z^2 \times N \times radix \times M^2}}{6 \times z \times radix \times M}$

Tabelle 7.2 zeigt die Ergebnisse, die bei der Einstellung des Parallelitätsgrades über das mathematische Modell für den Radix-Sortieralgorithmus erzielt wurden. Der Fehler zwischen den Ausführungszeiten, die einerseits durch den berechneten und andererseits den realen optimalen Parallelitätsgrad erzielt wurden, ist sowohl für eine Problemgröße von 64K mit als auch von 128K gering, nämlich kleiner als 2,5%. Obwohl die Abweichung zwischen dem berechneten und dem realen Parallelitätsgrad (Prozessoranzahl gleich 1 statt 2) gering bleibt, ist der Fehler bzgl. der Effizienz für diese Anwendung hoch (bis zu 96%). Dies ist auf die schlechte Skalierbarkeit des Programmes zurückzuführen, d.h. weil die Beschleunigung auf 2 Prozessoren niedrig ist, ist die Effizienz weit von dem optimalen Wert, der durch die Ausführung mit einem Prozessor erreicht wird. Der Fehler bei der Maximierung der Nutzen-Kosten-Relation beträgt in allen untersuchten Fällen Null.

Tabelle 7.2: Ergebnisse der Einstellung des Parallelitätsgrades für den Radix-Sortieralgorithmus, wenn $z = 4$ und $radix = 64K$

Radix-Sortieralgorithmus			
Problemgröße	Optimaler Parallelitätsgrad		Fehler
	Berechneter Wert	Realer Wert	
Ziel: Minimierung der Ausführungszeit			
64K, 4 Ziffer	2	4	0,49%
128K, 4 Ziffer	4	8	2,3%
Ziel: Einhaltung einer Effizienzschranke von 50%			
64K, 4 Ziffer	1	2	96,0%
128K, 4 Ziffer	1	2	88,6%
Ziel: Maximierung der Nutzen-Kosten-Relation			
64K, 4 Ziffer	1	1	0,0%
128K, 4 Ziffer	1	1	0,0%

7.2.3.3 LL13

Der Kommunikationsaufwand für *LL13* wächst linear (siehe Abschnitt 7.1.4). Die Ausführungszeit mit p Prozessoren (T_p) ist für *LL13* somit durch die folgende Formel gegeben:

$$T_p = \frac{T_1}{p} + \left(\frac{N}{p} + 64^2\right) \times niter \times M \tag{7.6}$$

wobei N und $niter$ die Problemgröße und M die Latenzzeit pro entfernten Zugriff (Kommunikationsoperation) darstellen. Die Effizienz in diesem Fall ist durch die Formel 7.2 und 7.6 im folgenden gegeben:

$$E_p = \frac{T_1}{T_1 + niter \times M \times N + 64^2 \times niter \times M \times p} \geq Effizienzschranke$$

Somit gilt $p \leq \frac{T_1 - E \times T_1 - E \times niter \times M \times N}{E \times 64^2 \times niter \times M}$, wobei E der Effizienzschranke entspricht.

Basierend auf den Formeln (7.3) und (7.6) ergibt sich für die Nutzen-Kosten-Relation:

$$R_{BC} = \frac{T_1 \times p}{(T_1 + N \times M + 64^2 \times p \times M)^2}$$

Die Nullstelle der ersten Ableitung für R_{BC} bzw. das Maximum der Nutzen-Kosten-Relation wird bei dem folgenden Parallelitätsgrad erreicht: $p = \frac{T_1 + N \times M}{64^2 \times M}$.

Tabelle 7.3 zeigt die Ergebnisse, die für die Einstellung des Parallelitätsgrades über das mathematische Modell für *LL13* erzielt wurden. Der Fehler zwischen der durch den berechneten und den realen optimalen Parallelitätsgrad erzielten Ausführungszeit, Effizienz bzw. Nutzen-Kosten-Relation bleibt für weniger umfangreiche Problemgrößen gering, nämlich im Durchschnitt kleiner als 7%. Für eine umfangreichere Problemgröße (32K) steigt der Fehler bis auf ca. 38%. Auch hier ist der Fehler vermutlich auf Cache-Effekte zurückzuführen (siehe Abschnitt 7.2.3.1).

Tabelle 7.3: Ergebnisse der Einstellung des Parallelitätsgrades für $LL13$, wenn $niter = 100$

Livermore Loop 13			
Problemgröße	Optimaler Parallelitätsgrad		Fehler
	Berechneter Wert	Realer Wert	
Ziel: Minimierung der Ausführungszeit			
4K	64	64	0,0%
16K	64	64	0,0%
32K	64	64	0,0%
Ziel: Einhaltung einer Effizienzschranke von 70%			
4K	2	4	13,7%
16K	2	8	15,7%
32K	2	8	17,1%
Ziel: Maximierung der Nutzen-Kosten-Relation			
4K	3	2	3,2%
16K	11	4	7,5%
32K	21	4	38,3%

7.2.3.4 TLM-Verfahren

Der Kommunikationsaufwand für das TLM-Verfahren wächst linear (siehe Abschnitt 7.1.6) mit der Problemgröße. Die Ausführungszeit mit p Prozessoren (T_p) ist für das TLM-Verfahren somit durch die folgende Formel gegeben:

$$T_p = \frac{T_1}{p} + (4 \times N \times niter \times M) \tag{7.7}$$

wobei $4 \times N \times niter$ die Gesamtanzahl der zu übertragenden Elemente und M die Latenzzeit pro entfernten Zugriff (Kommunikationsoperation) darstellen. Die Effizienz wird anhand der Formel 7.2 und 7.7 gegeben:

$$E_p = \frac{T_1}{T_1 + 4 \times niter \times M \times N \times p} \geq Effizienzschranke$$

Somit wird die Effizienzschranke bei $p = \frac{T_1 \times (1-E)}{E \times 4 \times niter \times M \times N}$ erreicht, wobei E die Effizienzschranke angibt.

Basierend auf den Formeln (7.3) und (7.7) wird die Nutzen-Kosten-Relation berechnet als:

$$R_{BC} = \frac{T_1 \times p}{(T_1 + 4 \times N \times niter \times M \times p)^2}$$

Die Nullstelle der ersten Ableitung für R_{BC} bzw. das Maximum der Nutzen-Kosten-Relation wird beim folgenden Parallelitätsgrad erreicht: $p = \frac{T_1}{4 \times N \times niter \times M}$

Tabelle 7.4: Ergebnisse der Einstellung des Parallelitätsgrades für das TLM-Verfahren, wenn $niter = 256$

TLM-Verfahren			
Problemgröße	Optimaler Parallelitätsgrad		Fehler
	Berechneter Wert	Realer Wert	
Ziel: Minimierung der Ausführungszeit			
12K	8	4	46,3%
48K	16	8	35,6%
Ziel: Einhaltung einer Effizienzschranke von 50%			
12K	2	4	14,9%
48K	2	4	8,6%
Ziel: Maximierung der Nutzen-Kosten-Relation			
12K	2	2	0,0%
48K	3	4	30,2%

Tabelle 7.4 zeigt die Ergebnisse der Einstellung des Parallelitätsgrades über das mathematische Modell. Für diese Anwendung spielt eine kleine Abweichung zwischen dem berechneten und dem realen Parallelitätsgrad besonders bei dem Ziel der Minimierung der Ausführungszeit eine entscheidende Rolle, da das Programm gut skaliert. Der Fehler beträgt deswegen für die kleinere Problemgröße im Durchschnitt ca. 20%. Wenn die Problemgröße vervierfacht wird, nimmt der Durchschnitt des Fehler um ca. 20% zu.

7.2.3.5 PDE-Löser

Der Kommunikationsaufwand für den PDE-Löser wächst quadratisch (siehe Abschnitt 7.1.7) mit der Anzahl der Prozessoren. Die Ausführungszeit mit p Prozessoren (T_p) ist daher gegeben durch die folgende Formel:

$$T_p = \frac{T_1}{p} + \left(\frac{Nn2}{p} \times Nn3^2 \times 2 + p^2 \right) \times M \times niter \tag{7.8}$$

M stellt die Latenzzeit pro entfernten Zugriff (Kommunikationsoperation) dar; $Nn2$, $Nn3$ bezeichnen die Problemgröße, $niter$ repräsentiert die Anzahl der Iterationen. Die Effizienz wird durch die Formel 7.2 und 7.8 ermittelt:

$$E_p = \frac{T_1}{T_1 + 2 \times M \times niter \times Nn2 \times Nn3^2 + M \times niter \times p^3} \geq Effizienzschranke$$

Die Effizienzschranke wird bei $p \leq \sqrt[3]{\frac{T_1 - E \times T_1 - E \times 2 \times M \times niter \times Nn2 \times Nn3^2}{E \times M \times niter}}$ erreicht, wobei E die Effizienzschranke angibt.

Basierend auf den Formeln (7.3) und (7.8) ergibt sich für die Nutzen-Kosten-Relation:

$$R_{BC} = \frac{T_1 \times p}{(T_1 + 2 \times M \times niter \times Nn2 \times Nn3^2 + M \times niter \times p^3)^2}$$

Hier wurde das Programm Maple [7] eingesetzt, um das Maximum der Nutzen-Kosten-Relation für zwei Problemgrößen zu finden.[3]

Tabelle 7.5: Ergebnisse der Einstellung des Parallelitätsgrades für den PDE-Löser, wenn $niter = 50$

PDE-Löser			
Problemgröße	Optimaler Parallelitätsgrad		Fehler
	Berechneter Wert	Realer Wert	
Ziel: Minimierung der Ausführungszeit			
16K	32	16	0,0%
32K	32	32	0,0%
Ziel: Einhaltung einer Effizienzschranke von 70%			
16K	2	16	21,0%
32K	2	16	21,0%
Ziel: Maximierung der Nutzen-Kosten-Relation			
16K	24	16	0,0%
32K	33	32	0,0%

Tabelle 7.5 zeigt die für den PDE-Löser erzielten Ergebnisse. Die hier verwendete Implementierung dieser Anwendung setzt eine Zweierpotenz von Prozessoren für die Ausführung voraus. Folge daraus ist, daß die berechneten optimalen Parallelitätsgrade weiter auf die letztmögliche Zweierpotenz von Prozessoren eingestellt werden müssen. Bei den untersuchten Problemgrößen entspricht dies einer Eistellung auf 16 bzw. 32 Prozessoren, wenn die Nutzen-Kosten-Relation betrachtet wird. Abgesehen von den Fällen, die die Einhaltung einer Effizienzschranke als Ziel haben, ist kein Fehler in den Ziel-Kennzahlenfestzustellen, der durch die Abweichung zwischen dem berechneten und dem realen optimalen Parallelitätsgrad zustande kommen würde.

7.2.3.6 LL3

Der Kommunikationsaufwand für LL3 wächst logarithmisch mit der Anzahl der Prozessoren (siehe Abschnitt 7.1.3). Die Ausführungszeit mit p Prozessoren (T_p) ist somit durch die folgende Formel gegeben:

$$T_p = \frac{T_1}{p} + (log(p) \times M) \tag{7.9}$$

M stellt die Latenzzeit pro entfernten Zugriff (Kommunikationsoperation) dar. Die Effizienz ist durch die Formel gegeben:

$$E_p = \frac{T_1}{T_1 + ln(p) \times M \times p} \geq Effizienzschranke$$

[3]Nachdem die Nutzen-Kosten-Relation R_{BC} abgeleitet wurde, wurden die Werte der bekannten T_1, M, $Nn1$, $Nn3$ und *niter* für die zwei Problemgrößen eingesetzt, wobei die Gleichung durch die Funktion *solve* von Maple numerisch gelöst wurde.

Auf Basis der Formeln (7.3) und (7.9) wird die Nutzen-Kosten-Relation wie folgt berechnet:

$$R_{BC} = \frac{T_1 \times p}{(T_1 + log(p) \times p \times M)^2}$$

Somit ist die erste Ableitung für die Nutzen-Kosten-Funktion: $R'_{BC} = \frac{T_1 \times (T_1 - ln(p) \times M \times p - 2 \times M \times p)}{(T_1 + ln(p) \times M \times p)^3}$.

Beim Vergleich zwischen den Ergebnissen eines gemessenen und eines über das mathematische Modell berechneten optimalen Parallelitätsgrades konnte für das Programm LL3 eine große Abweichung festgestellt werden. Bei einer kleinen Problemgröße (bis zu einem Virtualisierungsgrad von 512 Elementen) können alle Elemente im *First-Level-Cache* untergebracht werden. Auf der anderen Seite fällt die Kommunikationszeit in dieser Situation zu viel ins Gewicht, wenn die Anzahl der benutzten Prozessoren wächst. Außerdem sind die Prozessoren nicht gut ausgelastet. Bei einer umfangreicheren Problemgröße wird die Kommunikationszeit für LL3 im Vergleich zu der linearen Abnahme der Ausführungszeit nicht mehr wichtig, im *First-Level-Cache* ist aber nicht mehr Platz für alle Elemente. Folge daraus ist, daß *Cache-Effekte* zustande kommen, so daß die Modellierung der Berechnungs- bzw. der Kommunikationszeit als linear bzw. logarithmisch nicht mehr der Realität entspricht.[4]

Tabelle 7.6 zeigt die Entwicklung der Ausführungszeit sowie der Nutzen-Kosten-Relation anhand eines Beipiels, das eine Problemgröße von 32K Elementen berücksichtigt. Die erste Spalte gibt die Prozessoranzahl an. Die zweite und dritte Spalte zeigen die reale Ausführungszeit bzw. die reale Nutzen-Kosten-Relation, während die zwei darauffolgenden Spalten dasselbe präsentiert, berechnet jedoch über das mathematische Modell. Aus den Daten in der Tabelle kann man schließen, daß der reale optimale Parallelitätsgrad nach der maximalen Nutzen-Kosten-Relation 16 ist, wobei der optimale Parallelitätsgrad, der über das mathematische Modell berechnet wurde, 128 beträgt. Die Auswirkung dieser Abweichung ist das Erreichen einer Nutzen-Kosten-Relation (mit 128 Prozessoren gleich 1642,18), die um ca. Faktor 4 geringer ist als die maximale Nutzen-Kosten-Relation (mit 16 Prozessoren gleich 6702,80).

7.2.3.7 Veltran-Operator

Der Kommunikationsaufwand für den Veltran-Operator wächst linear mit dem Verhältnis zwischen der Anzahl der Elemente in der dreidimensionalen Matrix und der Anzahl der Prozessoren sowie logarithmisch mit der Anzahl der Prozessoren. Die Ausführungszeit mit p Prozessoren (T_p) ist:

$$T_p = \frac{T_1}{p} + \left(13 \times log(p) + \frac{x}{p} \times y \times z + \frac{x}{p}\left(x - \frac{x}{p}\right) \times y \times z + \frac{x}{p} \times y \times z\right) \times M \qquad (7.10)$$

wobei x, y bzw. z der Anzahl der Elemente der jeweiligen Dimension der Matrix und M der Latenzzeit pro entfernten Zugriff entsprechen. Die Effizienz wird durch die Formel 7.2 und 7.10 gegeben. Basierend auf den Formeln (7.3) und (7.10) erhält man für die Nutzen-Kosten-Relation:

[4] Hier bezieht sich die Kommunikationszeit sowohl auf die Inter- als auch Intra-Prozessor-Kommunikation. Mit Intra-Prozessor-Kommunikationszeit meint man hier die Kosten, die mit der Verfügbarkeit eines lokalen Elementes zu tun haben. Diese Kosten variieren je nachdem, woraus die Elemente herausgelesen werden müssen, nämlich aus dem *First-Level-Cache*, dem *Second-Level-Cache* oder aus dem Hauptspeicher.

Tabelle 7.6: Vergleich zwischen der realen und berechneten Ausführungszeit und Nutzen-Kosten-Relation für das Programm LL3 mit Problemgröße 32K

Livermore Loop 3				
	Real		Modell-basiert	
Parallelitätsgrad	Ausführungszeit(s)	Nutzen-Kosten	Ausführungszeit(s)	Nutzen-Kosten
1	0,002102	475,73	0,002102	475,06
2	0,001074	911,15	0,001052	948,79
4	0,000616	1384,80	0,000528	1881,64
8	0,000299	2939,00	0,000267	3680,21
16	0,000140	6702,80	0,000137	6968,30
32	0,000110	5428,70	0,000073	12291,87
64	0,000107	2868,70	0,000041	18850,00
128	0,000100	1642,18	0,000026	22859,14
256	0,000117	599,82	0,000020	20374,37

$$R_{BC} = \frac{T_1 \times ln^2(2) \times p^3}{(T_1 \times ln(2) \times p + M \times A)^2}$$

wobei $A = (13 \times ln(p) \times p^2 + 2 \times x \times y \times z \times ln(2) \times p + x^2 \times y \times z \times ln(2) \times p - x^2 \times y \times z \times ln(2)$.

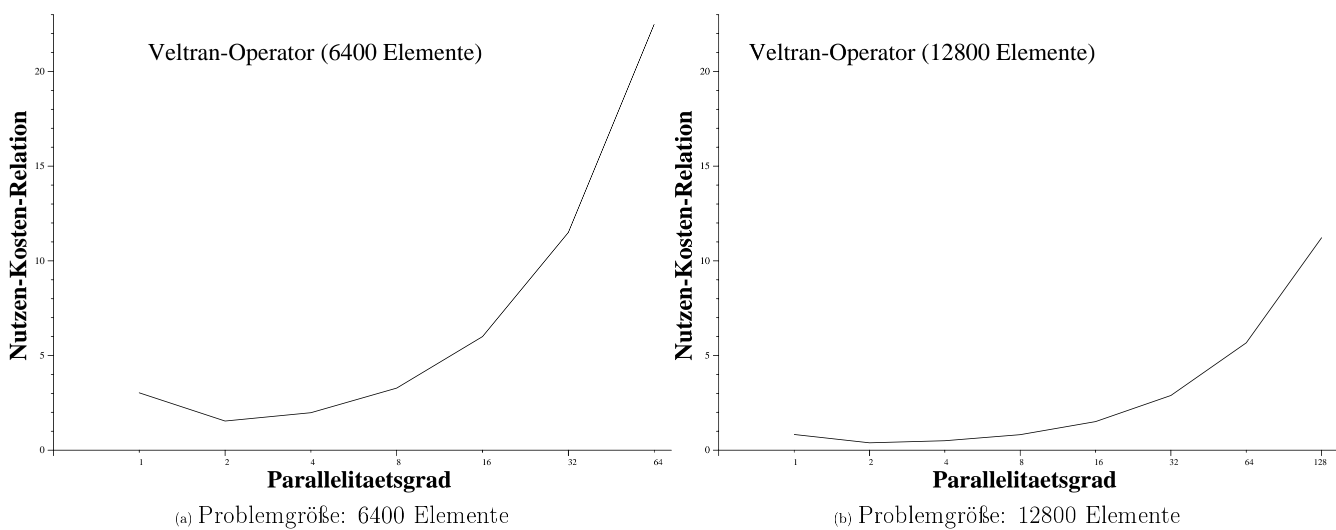
Die Kurven der Nutzen-Kosten-Relation für Problemgrößen von 6400 bzw. 12800 Elementen sind in Abbildungen 7.5a bzw. 7.5b dargestellt. Die maximale Prozessoranzahl ist für die angegebenen Problemgrößen jeweils 64 und 128, da der betrachtete Algorithmus voraussetzt, daß jeder Prozessor die Berechnung für mindestens eine Spalte durchführt (siehe Abschnitt 7.1.5). Anhand der Abbildung 7.5 kann man feststellen, daß die Nutzen-Kosten-Relation ihr Maximum beim Parallelitätsgrad 64 bzw. 128 erreicht.

Tabelle 7.7 zeigt die Ergebnisse, die bei der Einstellung des Parallelitätsgrades über das mathematische Modell für den Veltran-Operator erzielt wurden. Es gibt keine Abweichung zwischen den durch den berechneten und den realen optimalen Parallelitätsgrad erzielten Ausführungszeiten und Nutzen-Kosten-Relationen für eine Problemgröße von 6400 Elementen. Das gleiche Verhalten wiederholt sich, wenn sich die Problemgröße verdoppelt. Wenn das Ziel der Einstellung die Einhaltung einer Effizienzschranke von 90% ist, beträgt der Fehler im Durchschnitt ca. 18%.

7.2.4 Genauigkeit des mathematischen Modells

Bis auf einige Situationen, die in Abschnitt 7.2.3 beschrieben wurden, bleibt die Auswirkung der Abweichung zwischen dem auf der Cray T3E gemessenen und dem anhand der Ausführungszeit, der Effizienz bzw. der Nutzen-Kosten-Relation berechneten optimalen Parallelitätsgrad gering, nämlich im Durchschnitt ca. 13%. Daher kann man festellen, daß es sich lohnt, den Parallelitätsgrad über das mathematische Modell zu ermitteln.

Die Genauigkeit der durch das mathematische Modell gelieferten Ergebnisse hängt von zwei Faktoren ab: Erstens spielt die Abschätzung des Berechnungs- bzw. Kommunikationsaufwandes eines parallelen Programmes eine signifikante Rolle. Zweitens beeinflussen



(a) Problemgröße: 6400 Elemente

(b) Problemgröße: 12800 Elemente

Abbildung 7.5: Entwicklung der Nutzen-Kosten-Relation für den Veltran-Operator nach dem mathematischen Modell

Tabelle 7.7: Ergebnisse der Einstellung des Parallelitätsgrades für den Veltran-Operator

Veltran-Operator			
Problemgröße	Optimaler Parallelitätsgrad		Fehler
	Berechneter Wert	Realer Wert	
Ziel: Minimierung der Ausführungszeit			
6400	64	64	0,0%
12800	128	128	0,0%
Ziel: Einhaltung einer Effizienzschranke von 90%			
6400	16	2	20,0%
12800	32	2	17,9%
Ziel: Maximierung der Nutzen-Kosten-Relation			
6400	64	64	0,0%
12800	128	128	0,0%

Cache-Effekte stark die Abweichung zwischen dem berechneten und gemessenen optimalen Parallelitätsgrad. Durch Cache-Effekte kann die Abschätzung für den Berechnungs- bzw. Kommunikationsaufwand mißgelingen, wie für das Programm LL3 im Abschnitt 7.2.3.6 beschrieben wurde.

7.2.5 Extrapolation aus den erzielten Ergebnissen

Die im mathematischen Modell definierten Klassen (siehe Abschnitt 4.2), die Programme nach ihrem Kommunikationsverhalten gruppieren, umfassen ein breites Spektrum von parallelen Programmen. Die auf das mathematische Modell hin untersuchte Benchmark-Sammlung besteht aus mindestens einem Programm aus jeder Klasse. Aus den durch die

Anwendung des mathematischen Modells erzielten Ergebnissen kann extrapoliert werden, daß der optimale Parallelitätsgrad eines jeden weiteren Programmes, das sich zu einer dieser Klassen zuordnen läßt, anhand des mathematischen Modells berechnet werden kann. Außerdem können für solche Programme, deren Kommunikationsverhalten außerhalb der spezifizierten Klassen liegen, neue Klassen definiert werden und somit der optimale Parallelitätsgrad nach der Nutzen-Kosten-Relation bestimmt werden. Die Grundzüge des Verhaltens der Ausführungszeit sowie der Nutzen-Kosten-Relation und dadurch die Bestimmung des optimalen Parallelitätsgrades sind aus der vorliegenden Arbeit auch dann zu entnehmen, wenn das Kommunikationsverhalten eines parallelen Programmes einer Kombination der hier vorgeschlagenen Klassen entspricht.

Das mathematische Modell zur Bestimmung des optimalen Parallelitätsgrades ist außerdem auf alle Parallelrechner übertragbar.

7.3 Einstellung über Suchverfahren

Obwohl das mathematische Modell so gute Ergebnisse liefert, ist es nur bedingt in der Praxis anzuwenden, nämlich für solche Programme, für die sowohl der Berechnungs- als auch der Kommunikationsaufwand vorliegen. Solch eine Einschränkung ist bei der Benutzung der Einstellung des Parallelitätsgrades über Suchverfahren nicht vorhanden. Die Ergebnisse, die sich auf die Bestimmung des Parallelitätsgrades über Suchverfahren beziehen, wurden durch den Einsatz des in Kapitel 6 vorgestellten Systems namens ParGrad erzielt. Im folgenden werden die bei der Zusammenstellung der Ergebnisse eingesetzte Validierungstechnik präsentiert und die erzielten Ergebnisse diskutiert. Dieser Abschnitt beschäftigt sich außerdem mit der Analyse der Genauigkeit des Suchverfahrens sowie mit der Extrapolation aus den erzielten Ergebnissen.

7.3.1 Validierungstechnik

Um die Einstellung des Parallelitätsgrades von Programmen über Suchverfahren zu untersuchen, wurde das System ParGrad (siehe Kapitel 6) im Rahmen dieser Arbeit entwickelt. Die in der Benchmark-Sammlung enthaltenen Programme wurden manuell instrumentiert (siehe Abschnitt 6.7) und unter ParGrad ausgeführt.

Die Messungen wurden auf der Cray T3E/900 der Abteilung *Höchstleistungsrechnen - Stuttgart* (HLRS) der Universität Stuttgart durchgeführt. Der Parallelrechner Cray T3E [46, 41] der Universität Stuttgart besteht aus 512 DEC-Alpha-21164-Prozessoren, deren Taktrate jeweils 450 MHz beträgt. Die Prozessoren sind über ein hochperformantes, bidirektionales 3D-Torus-Netz verbunden. Das massiv-parallele Prozessorsystem Cray T3E ist ein skalierbares MIMD-(**M***ultiple* **I***nstruction* **M***ultiple* **D***ata*) System mit gemeinsamem Adreßraum (siehe Abschnitt 2.2). Der lokale Speicher beträgt 128 MB pro Prozessor. Das Betriebssystem ist das UNICOS/mk Version 2.0.4.43, das auf dem UNIX System V von UNIX System Laboratories basiert.

Bei der Entwicklung der Benchmark-Sammlung wurde die Cray-*shmem*-Bibliothek (siehe Abschnitt 6.3) für die Inter-Prozessor-Kommunikation eingesetzt.

7.3.2 Mehraufwand der Instrumentierung

Die Instrumentierung paralleler Programme zur Benutzung des ParGrad-Systems erzeugt einen geringen Mehraufwand, d.h. die parallelen Programme müssen nur geringfügig mo-

difiziert werden. Tabelle 7.8 zeigt die Anzahl der Zeilen, die für die jeweilige Anwendung des ParGrad-Systems hinzugefügt werden mußten, sowie den entsprechenden prozentualen Mehraufwand im Vergleich zum nicht-instrumentierten Programm. In Tabelle 7.8 ist der Mehraufwand für vier Programme dargestellt, wobei nur Versionen der Programme mit einer einzigen Einstellungsumgebung betrachtet wurden. Auch für kleinere Programme, wie z.B. LL1 und LL3, hält sich der Mehraufwand der Instrumentierung in Grenzen, nämlich bis zu ca. 13%.

Tabelle 7.8: Mehraufwand bei der Instrumentierung von 4 Programmen

Program	hinzugefügte Zeilen	Mehraufwand (%)
Radix-Sortieralgorithmus	20	3,1
PDE-Löser	32	2,3
Veltran-Operator	25	3,3
TLM-Verfahren	33	2,6
Livermore Loop 1	18	10,5
Livermore Loop 3	18	12,8
Livermore Loop 13	17	6,0

7.3.3 Zur Präsentation der Ergebnisse

Die über das Suchverfahren erzielten Ergebnisse werden im folgenden graphisch dargestellt. Die Ergebnisse der Ausführungszeit werden mit Hilfe von Säulen-Diagrammen vorgestellt, während die Ergebnisse der Effizienz sowie der Nutzen-Kosten-Relation als Kurven aufgeführt werden.

Abbildung 7.6 zeigt die Schablone für die graphische Präsentation der Ergebnisse bzgl. der Ausführungszeit. Die Diagrammart auf der *linken Seite* wird für die Präsentation der erzielten Ergebnisse für das jeweils betrachtete parallele Programm *ohne Einstellung des Parallelitätsgrades* verwendet. Die x-Achse stellt die Anzahl der Prozessoren dar, die jeweils während der gesamten Ausführungszeit des Programmes benutzt wurde. Jede Säule stellt eine Ausführung des Programmes dar.

Die Diagrammart auf der *rechten Seite* bezieht sich auf Ausführungen des Programmes *mit der Einstellung des Parallelitätsgrades*. In diesem Fall stellt die x-Achse die jeweilige maximale Anzahl von Prozessoren dar, die bei einer der Ausführung des Programmes zur Verfügung steht.

In dem Diagramm, das die Ausführung des parallelen Programmen mit der Einstellung des Parallelitätsgrades betrachtet, gibt jede Säule die Zeit einer Ausführung des parallelen Programmes wieder, wenn eine Intra-Lauf-Einstellung vorgenommen wird, bzw. die Gesamtzeit mehrerer Ausführungen, falls es sich um eine Inter-Lauf-Einstellung handelt. Eine Säule besteht aus mehreren Segmenten, wobei jedes Segment die Ausführungszeit eines Teiles des Programmes mit der entsprechenden Anzahl von Prozessoren für eine Intra-Lauf-Einstellung darstellt, bzw. die Ausführungszeit des gesamten Programmes, falls eine Inter-Lauf-Einstellung vorgenommen wird. In einem Segment repräsentiert die Zahl in Klammern die Anzahl der tatsächlich benutzten Prozessoren bei der Ausführung eines Teiles des Programmes (Intra-Lauf), bzw. während einer kompletten Ausführung des Programmes (Inter-

Lauf). Die Zahlen oberhalb der Säulen, die in der Abbildung auf der rechten Seite zu sehen sind, stellen die Variation des Parallelitätsgrades in der Zeit dar, d.h. das Parallelitätsprofil, wobei der zuletzt abgebildete Parallelitätsgrad derjenige ist, der durch die jeweils betrachtete Strategie als Optimum ausgewählt wurde.

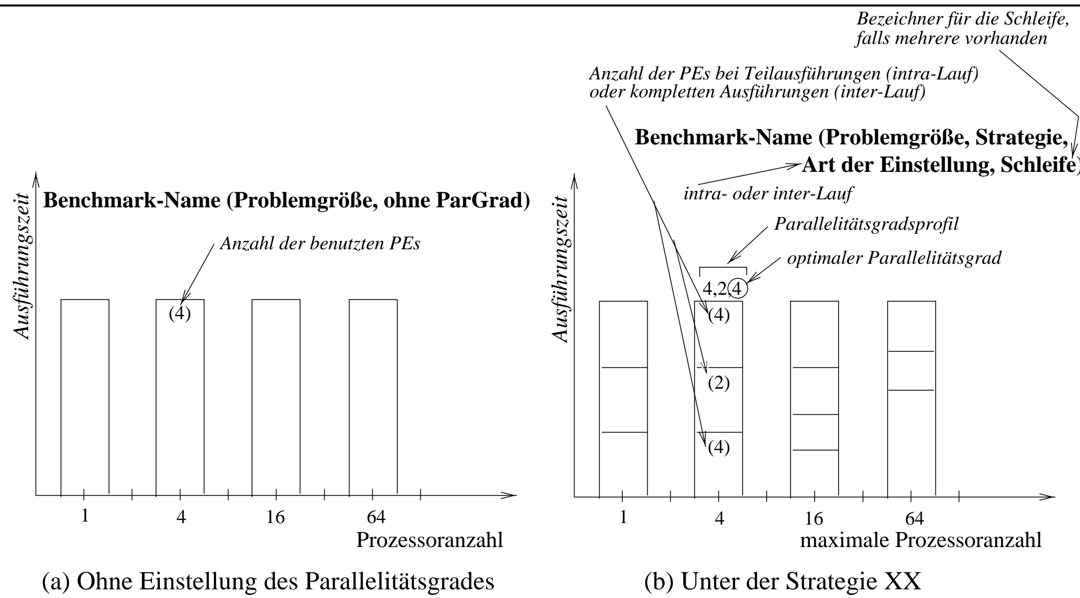


Abbildung 7.6: Schablone für die graphische Präsentation der Ergebnisse bzgl. der Ausführungszeit

Abbildung 7.7 stellt die Schablone für die graphische Präsentation der Ergebnisse bzgl. der Kennzahlen Effizienz und Nutzen-Kosten-Relation dar. Das Diagramm auf der *linken Seite* zeigt die Entwicklung der entsprechenden Kennzahl in Abhängigkeit von der Prozessoranzahl *ohne Einstellung des Parallelitätsgrades. In der Mitte* wird das Diagramm vorgestellt, das die Entwicklung der Kennzahl abhängig von der maximalen Prozessoranzahl *während der Einstellung des Parallelitätsgrades* wiedergibt. Bei einer bestimmten maximalen Prozessoranzahl wird der Parallelitätsgrad über mehrere Iterationen eingestellt. Die in einer Iteration eingesetzte Prozessoranzahl kann von der maximalen Prozessoranzahl abweichen. Zudem ist die Prozessoranzahl von Iteration zu Iteration nicht unbedingt gleich. Bei der Berechnung der Kennzahl für eine bestimmte maximale Prozessoranzahl wird ein Mittelwert über alle Iterationen gebildet, wobei die jeweils tatsächlich benutzte Prozessoranzahl maßgeblich ist. Das Diagramm auf der *rechten Seite* zeigt die Entwicklung der entsprechenden Kennzahl in Abhängigkeit von der maximalen Prozessoranzahl bei einer Ausführung des parallelen Programmes *nach der Einstellung des Parallelitätsgrades*, d.h. wenn der optimale Parallelitätsgrad bereits bekannt ist.

7.3.4 Diskussion der Ergebnisse

Hier werden die Ergebnisse der Einstellung des Parallelitätsgrades über Suchverfahren (siehe Kapitel 5) für jedes Programm der oben präsentierten Benchmark-Sammlung diskutiert. Dabei werden u.a. die betrachtete Problemgröße, das Ziel der Einstellung, die umzuverteilenden Datenstrukturen und die Art der Einstellung vorgestellt.

Bei der Zusammenstellung der Ergebnisse wurden die folgenden Fragen untersucht:

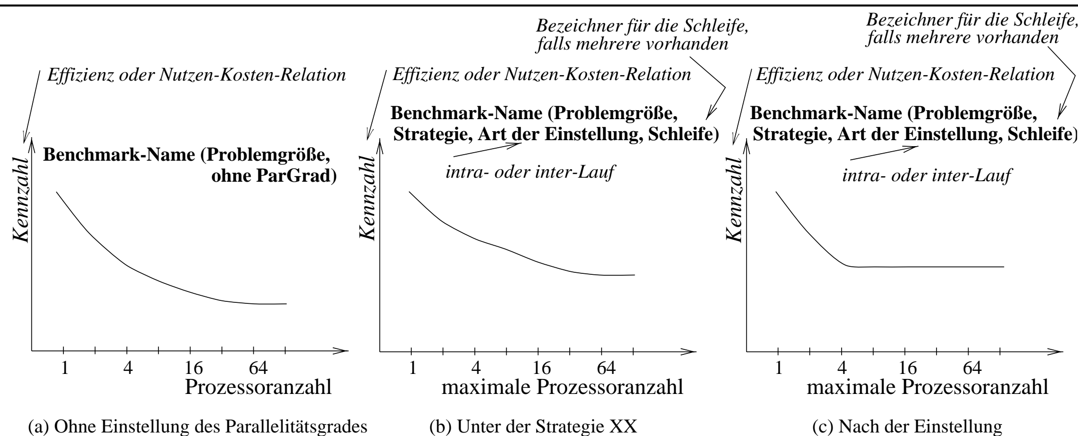


Abbildung 7.7: Schablone für die graphische Präsentation der Ergebnisse bzgl. der Effizienz und der Nutzen-Kosten-Relation

1. Wieviel schneller ist die Ausführung des Programmes unter *ParGrad* im Vergleich zu einer Ausführung ohne Einstellung des Parallelitätsgrades?

2. Wieviel effizienter ist die Ausführung eines Programmes unter *ParGrad* im Vergleich zu einer Ausführung ohne Einstellung des Parallelitätsgrades?

3. Wieviel höher ist die Nutzen-Kosten-Relation eines Programmes unter *ParGrad* im Vergleich zu einer Ausführung ohne Einstellung des Parallelitätsgrades?

4. Wie hoch ist der Mehraufwand für die Einstellung des Parallelitätsgrades, d.h. für die strategiespezifischen Messungen und für die Datenumverteilung?

5. Wurde das festgelegte Ziel für die Einstellung erreicht?

6. Haben die verschiedenen Strategien in solchen Fällen, in denen das Ziel der Einstellung während der Ausführung des Programmes nicht erreicht werden konnte, trotzdem das richtige Ergebnis bzw. den optimalen Parallelitätsgrad geliefert?

Zu Anfang der Diskussion jeder untersuchten Anwendung zur Einstellung des Parallelitätsgrades werden die Ergebnisse mit Hilfe der oben erläuterten Fragen tabellarisch zusammengefaßt. Die erste Spalte der Tabelle gibt das Ziel der Einstellung an, gefolgt durch die Strategie und die Art der Einstellung. Die fünfte Spalte bezieht sich auf den Mehraufwand der Einstellung des Parallelitätsgrades, wobei dieser auf dem schlimmsten Fall beruht. Bei einer Inter-Lauf-Einstellung bezieht sich der Mehraufwand auf die Messungen, während er sich bei einer Intra-Lauf-Einstellung auf die Kombination der Messungen und der benötigten Datenumverteilung bezieht. Danach wird angegeben, ob das festgelegte Ziel erreicht wurde, d.h. ob die Ausführungszeit während der Einstellung ihr Minimum erzielt hat, die Effizienz nicht unterhalb der gewünschten Schranke lag oder die Nutzen-Kosten-Relation ihren maximalen Wert erreicht hat. Die siebte Spalte gibt wieder, ob die verschiedenen Strategien den optimalen Parallelitätsgrad auch in dem Fall liefern konnten, wenn das festgelegte Ziel nicht erreicht wurde. Die letzte Spalte stellt die Auswirkung der Einstellung des Parallelitätsgrades dar. Diese Auswirkung beruht, je nach Ziel der Einstellung, auf die Ausführungszeit, Effizienz oder Nutzen-Kosten-Relation. Die tabellarische Darstellung der Ergebnisse wird für die erste Anwendung im nächsten Abschnitt detailliert erläutert.

7.3.4.1 Radix-Sortieralgorithmus

Die für den Radix-Sortieralgorithmus erzielten Ergebnisse sind in der Tabelle 7.9 zusammengefaßt. In allen dargestellten Fällen gibt es eine einzige Einstellungsumgebung, d.h. die Einstellung des Parallelitätsgrades findet nur an einer Stelle im Programm statt. Die Datenstruktur, die wegen einer Änderung im Parallelitätsgrad umverteilt werden muß, ist für das Radix-Programm das Feld, das die zu sortierenden Elemente enthält.

Für alle untersuchten Strategien blieben die Kosten für die Messungen und für die Datenumverteilung kleiner als 1% der Gesamtausführungszeit des Radix-Programmes.

Tabelle 7.9: Ergebnisse der Einstellung des Parallelitätsgrades für den Radix-Sortieralgorithmus

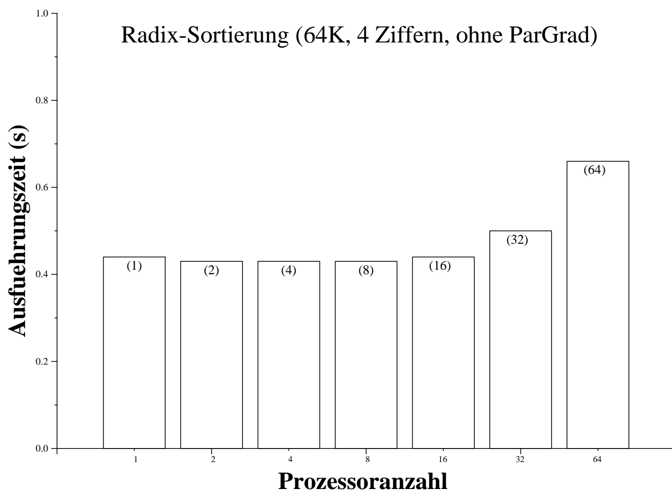
Radix-Sortieralgorithmus						
Einstellung				Ziel	Optimum	Auswirkung
Ziel	Strategie	Art	Mehraufwand	erreicht ?	gefunden?	
Ausführungszeit	RISS	intra-Lauf	$< 1\%$	nein	ja	− Faktor 3,5
Ausführungszeit	PeSS	inter-Lauf	$< 1\%$	ja	ja	− Faktor 3,5
Effizienz	EISS	intra-Lauf	$< 1\%$	nein	ja	+ Faktor 20
Nutzen-Kosten	CBISS	intra-Lauf	$< 1\%$	nein	ja	+ Faktor 26

Hier wird im Rahmen eines Beispiels erläutert, wie Tabelle 7.9 gelesen werden soll: Das Ziel der ersten Einstellung ist die Minimierung der Ausführungszeit. Dafür wurde die Strategie RISS angewendet, wobei eine Intra-Lauf-Einstellungen vorgenommen wurde. Der Mehraufwand der Einstellung blieb unter 1% der Gesamtausführungszeit. Die Ausführungszeit hat ihr Minimum unter der Strategie RISS zwar nicht erreicht, die Korrektheit der Einstellung konnte aber festgestellt werden, d.h. der Parallelitätsgrad wurde auf die Prozessoranzahl eingestellt, die zur Minimierung der Ausführungszeit führt. Die Auswirkung des Ergebnisses bzw. der Einstellung auf den optimalen Parallelitätsgrades war die Reduzierung der Ausführungszeit bis zum Faktor 3,5 (siehe Abbildung 7.11).

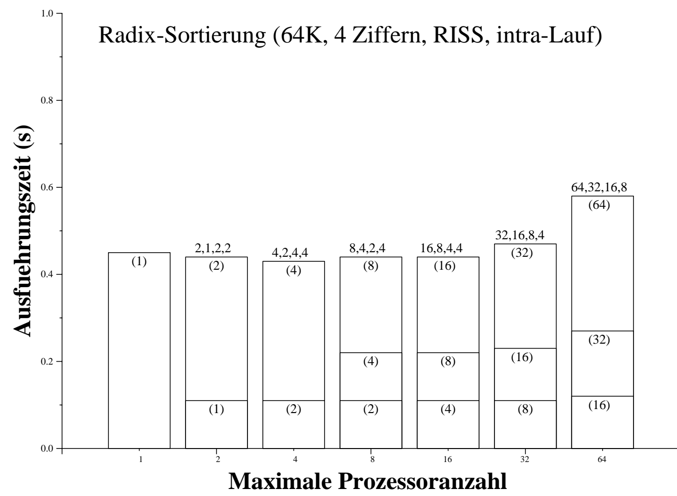
Einstellung Ausführungszeit: Der Parallelitätsgrad wurde hier mit dem Ziel eingestellt, die Ausführungszeit des Programmes zu minimieren. Darauf basierend wurden die Strategien RISS und PeSS (siehe Abschnitt 5.5.1) angewendet.

Die Ausführungunszeit des Radix-Programmes für eine Problemgröße gleich 64K Elemente und 4 Ziffern ($radix = 64K$) pro Schlüssel war unter der Benutzung der RISS-Strategie (siehe Abbildung 7.8b) bis zu 20% geringer als ohne Einstellung (siehe Abbildung 7.8a). Dabei wurde der optimale Parallelitätsgrad nach der Strategie RISS richtig auf 4 eingestellt. Wenn die Ausführung mit 64 Prozessoren anfängt, wird die Einstellung bei einem Parallelitätsgrad gleich 8 anstatt 4 beendet, weil das Programm vor der Einstellung auf 4 zu Ende geht. Bei einer Inter-Lauf-Einstellung des Parallelitätsgrades (siehe Abbildung 7.9) kann für diesen Fall festgestellt werden, daß die Einstellung, wie erwartet, auf 4 erfolgt.

Bei der Inter-Lauf-Einstellung über 8 Iterationen unter der Benutzung der RISS-Strategie wurde der Parallelitätsgrad auf 4 eingestellt, wobei dies in Abbildung 7.9b mit der Säule, die am weitesten rechts liegt, gezeigt ist. Dies ist der Punkt, an dem die Ausführungszeit am geringsten war. Bei einer Inter-Lauf-Einstellung ist eine Datenumverteilung nicht

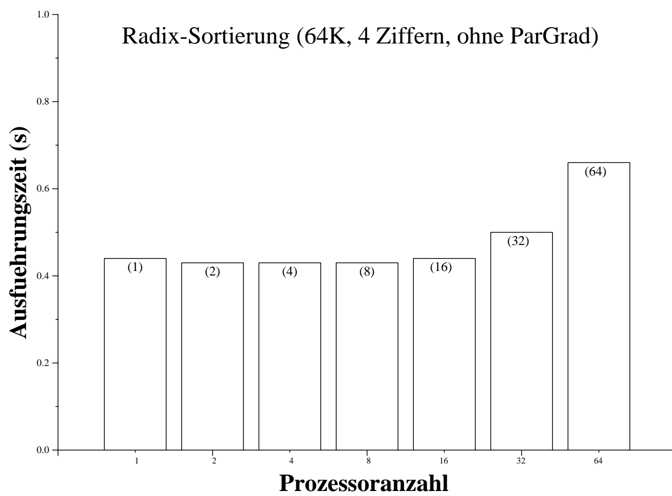


(a) Ohne Einstellung des Parallelitätsgrades

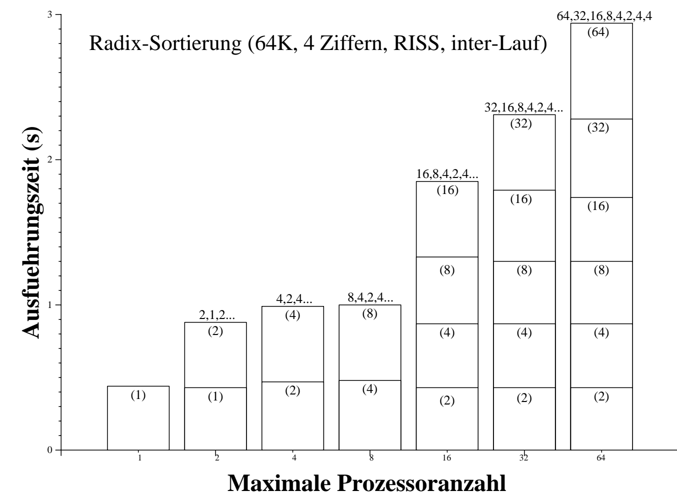


(b) Unter der Strategie RISS

Abbildung 7.8: Radix-Sortierung unter der RISS-Strategie mit Intra-Lauf-Einstellung



(a) Ohne Einstellung des Parallelitätsgrades

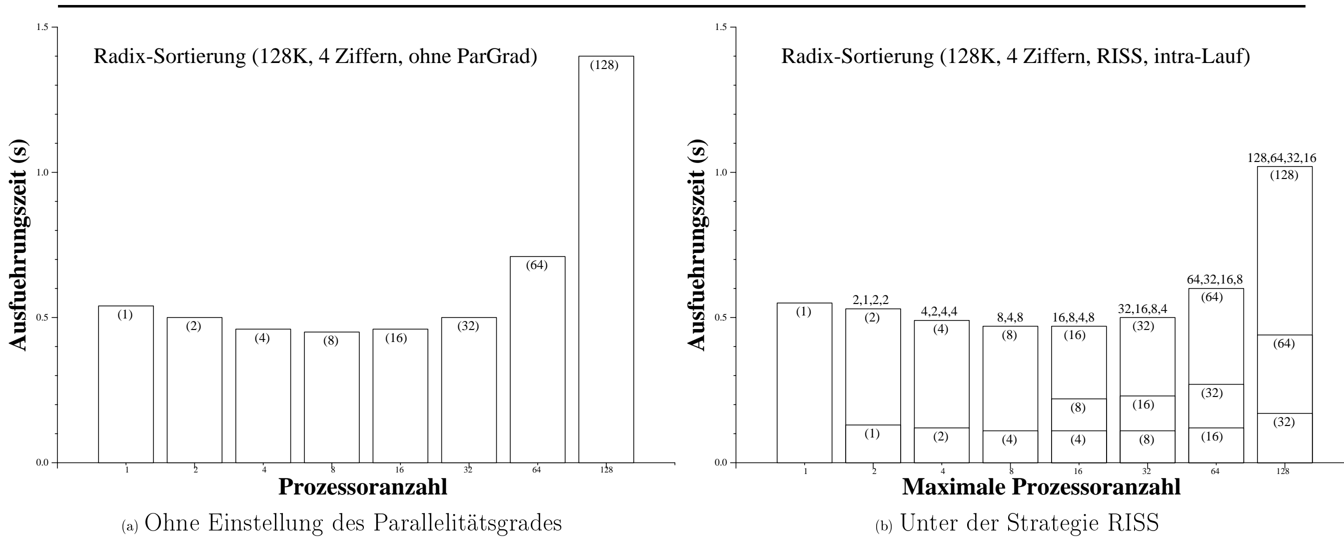


(b) Unter der Strategie RISS

Abbildung 7.9: Radix-Sortierung unter der RISS-Strategie mit Inter-Lauf-Einstellung

nötig; der Mehraufwand für die Einstellung des Parallelitätsgrades betrug weniger als 1% der Gesamtlaufzeit.

Abbildung 7.10 zeigt die Einstellung des Parallelitästgrades für das Radix-Program skaliert auf Problemgröße 128K unter der Benutzung der Strategie RISS. Die Ausführungunszeit war bis zu 36% geringer als ohne Einstellung (siehe Abbildung 7.10a und b). Dabei war der Parallelitätsgrad nach der Strategie RISS richtig auf 8 eingestellt (siehe Abbildung 7.10a). Wenn die Ausführung mit 128 Prozessoren anfängt, wird die Einstellung bei einem Parallelitätsgrad gleich 16 anstatt 8 beendet, weil das Programm vor der Einstellung auf 8 zu Ende geht.



(a) Ohne Einstellung des Parallelitätsgrades

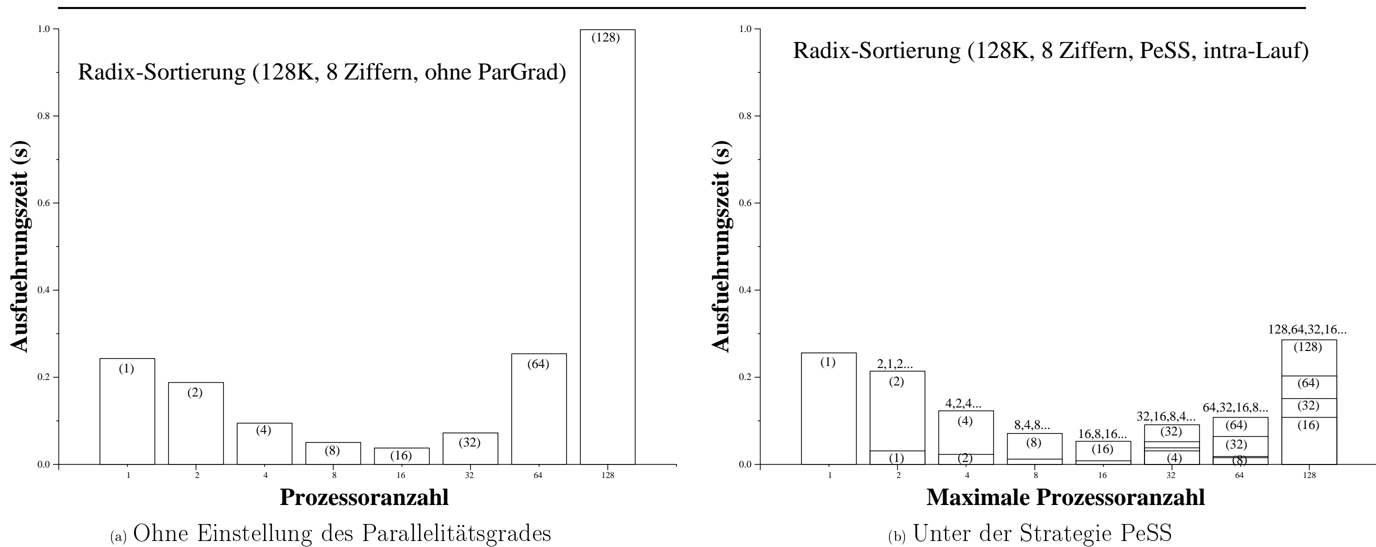
(b) Unter der Strategie RISS

Abbildung 7.10: Radix-Sortierung unter der RISS-Strategie mit Intra-Lauf-Einstellung

Noch weiterhin mit demselben Ziel, d.h. der Reduktion der Ausführungszeit des Radix-Programmes, wurde der Parallelitätsgrad nach der Strategie PeSS eingestellt. Abbildung 7.11 zeigt einen analysierten Fall, nämlich mit Problemgröße gleich 128K Elementen und acht Ziffern ($radix = 256$) pro Schlüssel. Unter der Strategie PeSS wurde die richtige Entscheidung für die drei Fälle mit einer maximalen Prozessoranzahl von 32, 64 128 getroffen, nämlich der Parallelitätsgrad wurde ab einem Punkt nicht weiter reduziert, weil dies zu einer Erhöhung der Ausführungszeit geführt hätte. Für diese drei Fälle hat die Strategie PeSS zwei Einstellungsschritte weniger gebraucht im Vergleich zu der Strategie RISS. Beispielsweise für eine maximale Prozessoranzahl gleich 128 muß der Parallelitätsgrad nur auf 64, 32 bzw. 16 heruntergesetzt werden (siehe Abbildung 7.11b), wobei für die Strategie RISS der Parallelitätsgrad auf 64, 32, 16, 8 bzw. 16 schrittweise eingestellt werden würde. Für eine maximale Prozessoranzahl von 1 bis 16 wird der Parallelitätsgrad wie bei der Strategie RISS eingestellt, da die Anzahl der möglichen Parallelitätsgrade 4 nicht überschreitet. 4 stellt die Anzahl der erforderlichen Meßpunkte dar, auf denen die Vorhersage der Strategie PeSS (siehe Abschnitt 5.5.1) basiert.

Die Ausführungszeit des Programmes wurde unter der Benutzung der Strategie PeSS bis zum Faktor 3,5 geringer als bei der Ausführung ohne Einstellung des Parallelitätsgrades. Dies ist in Abbildung 7.11b bei einer maximalen Prozessoranzahl gleich 128 im Vergleich zu 7.11a zu sehen. Bei einem initialen Parallelitätsgrad gleich 64 wird die Ausführungszeit um Faktor 2,4 niedriger im Vergleich zur Ausführungszeit ohne Einstellung des Parallelitätsgrades. Bei einer maximalen Prozessoranzahl gleich 4, 8, 16 oder 32 wird die Ausführungszeit aber höher,

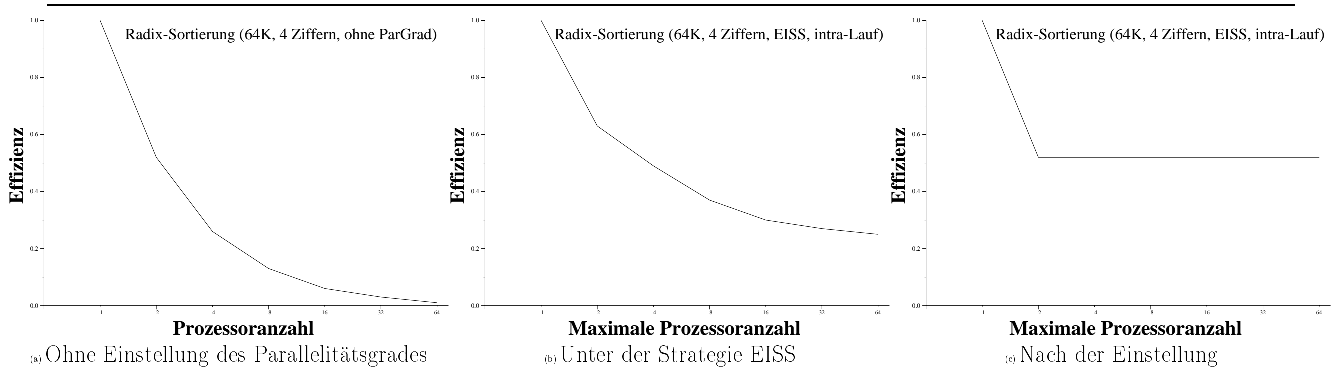
weil die Kosten der Einstellung im Vergleich zu der Reduzierung in der Ausführungszeit zu hoch sind.



(a) Ohne Einstellung des Parallelitätsgrades (b) Unter der Strategie PeSS

Abbildung 7.11: Radix-Sortierung unter der PeSS-Strategie mit Intra-Lauf-Einstellung

Einstellung Effizienz: Der Parallelitätsgrad wurde hier mit dem Ziel eingestellt, die Effizienz des Programmes zu steigern, und dadurch die Auslastung des Gesamtsystems zu verbessern. Darauf basierend wurde die Strategie EISS (siehe Abschnitt 5.5.1) angewendet.



(a) Ohne Einstellung des Parallelitätsgrades (b) Unter der Strategie EISS (c) Nach der Einstellung

Abbildung 7.12: Radix-Sortierung unter der EISS-Strategie mit Intra-Lauf-Einstellung

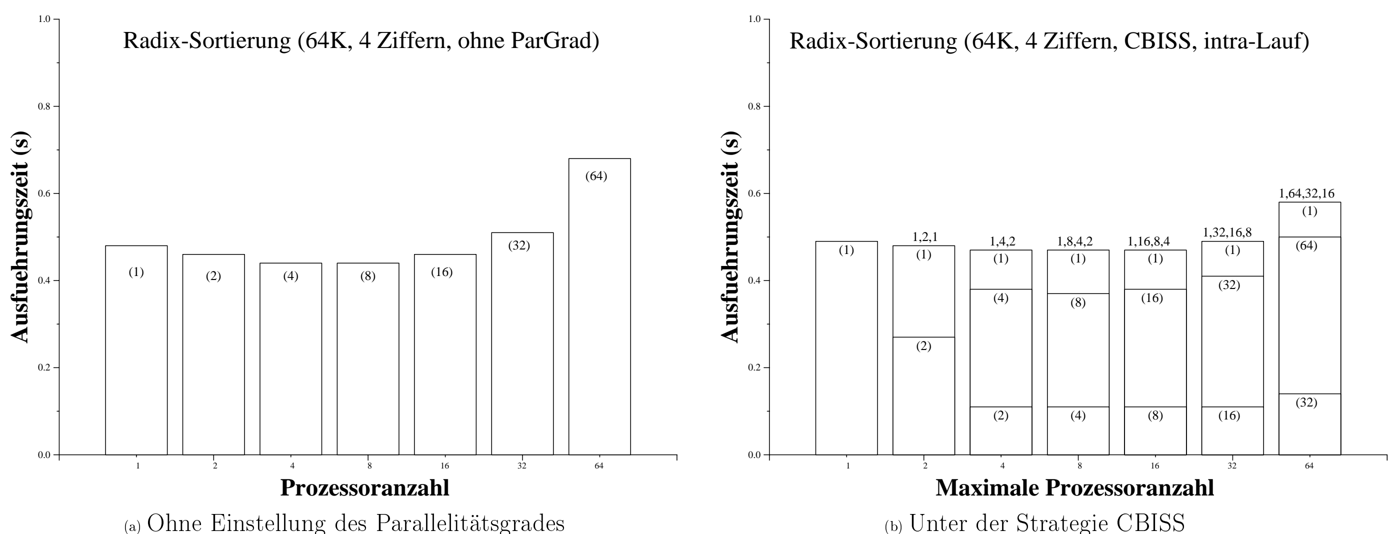
Im Vergleich zu einer Ausführung des Programmes ohne Einstellung des Parallelitätsgrades lief das Programm für eine Problemgröße gleich 64K z.B. mit 8 Prozessoren 6,2% langsamer unter der Benutzung der EISS-Strategie, wobei die Effizienzschranke 50% betrug. Der Parallelitätsgrad wurde dabei auf 2 eingestellt. Die Effizienz ist bereits während der Einstellung bis zum Faktor 20 gestiegen, wobei die Ausführungszeit wegen der Eistellung bis zu 6,3% erhöht wurde. Abbildung 7.12 zeigt einen Vergleich zwischen der Entwicklung der Effizienz für eine Ausführung ohne Einstellung des Parallelitätsgrades, während der Einstellung und nach der Einstellung, d.h. wenn der Parallelitätsgrad schon bekannt war. Im letzten Fall wurde der optimale Parallelitätsgrad aus der Parallelitätsgrad-Datenbank (siehe 5.9) ge-

lesen. Für alle untersuchten Fälle ist die Effizienz mit der Einstellung des Parallelitätsgrades höher geworden im Vergleich zu der Situation ohne Einstellung.

Mit einer maximalen Prozessoranzahl gleich 64 lief das Programm sogar 17% schneller unter der Benutzung der EISS-Strategie als ohne Einstellung, wobei der Parallelitätsgrad auf 16 eingestellt wurde (die Ausführung des Programmes war zu Ende bevor der Parallelitätsgrad auf 2 eingestellt werden konnte).

Einstellung Nutzen-Kosten: Der Parallelitätsgrad wurde hier eingestellt, um die Nutzen-Kosten-Relation des Programmes zu maximieren. Dafür wurde die Strategie CBISS (siehe Abschnitt 5.5.1) angewendet.

Die maximale Nutzen-Kosten-Relation wird in diesem Fall bei einem Parallelitätsgrad gleich 1 erreicht. Unter der Benutzung der Strategie CBISS wurde der Parallelitätsgrad immer weiter reduziert, bis entweder die Anzahl der benutzten Prozessoren bei 1 lag oder die Ausführung des Programmes zu Ende war. Dabei konnte eine Erhöhung der Ausführungszeit von bis zu 6,5% festgestellt werden, wie Abbildung 7.13 zeigt. Für eine maximale Prozessoranzahl gleich 32 bzw. 64 ist sogar die Ausführungszeit bis zu 17% gesunken im Vergleich zu der Ausführungszeit des Programmes ohne Einstellung. Die Nutzen-Kosten-Relation wurde bereits während der Einstellung bis zum Faktor 26 höher, wie Abbildung 7.14 zeigt.



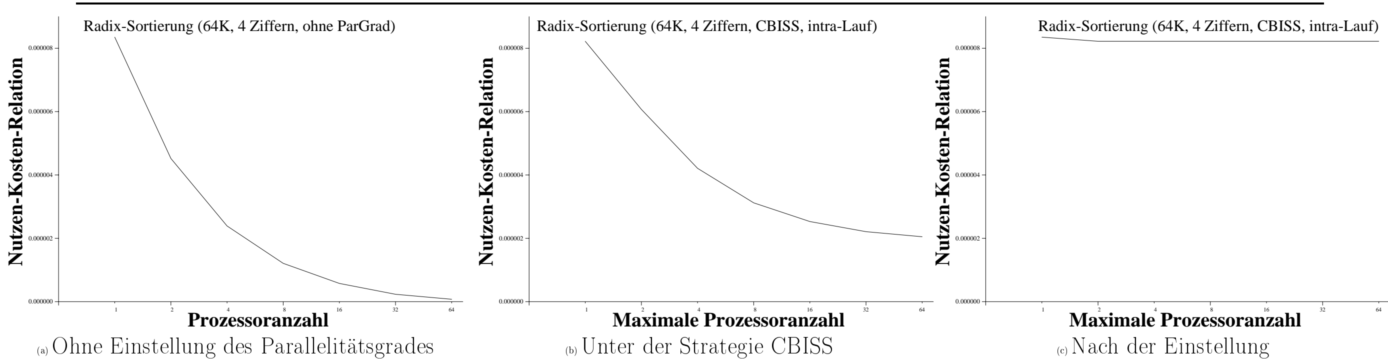
(a) Ohne Einstellung des Parallelitätsgrades (b) Unter der Strategie CBISS

Abbildung 7.13: Radix-Sortierung unter der CBISS-Strategie mit Intra-Lauf-Einstellung

7.3.4.2 *Livermore Loop 1* (LL1)

Hier werden die für das Programm *Livermore Loop 1* erzielten Ergebnisse diskutiert. Die Ergebnisse sind in der Tabelle 7.10 zusammengefaßt dargestellt. Jede betrachtete Einstellung hat eine einzige Einstellungsumgebung. Die Datenstrukturen, die wegen einer Änderung des Parallelitätsgrades umverteilt werden müssen, sind zwei eindimensionale Felder, die Eingabedaten bzw. partielle Ergebnisse enthalten.

Einstellung Ausführungszeit: Hier wird die Minimierung der Ausführungszeit des Programmes LL1 als Ziel verfolgt. Abbildung 7.15 zeigt das Ergebnis einer Intra-Lauf-Einstellung des Parallelitätsgrades unter der Strategie RISS, welche den Parallelitätsgrad



(a) Ohne Einstellung des Parallelitätsgrades (b) Unter der Strategie CBISS (c) Nach der Einstellung

Abbildung 7.14: Entwicklung der Nutzen-Kosten-Relation der Radix-Sortierung unter der CBISS-Strategie mit Intra-Lauf-Einstellung

Tabelle 7.10: Ergebnisse der Einstellung des Parallelitätsgrades für LL1

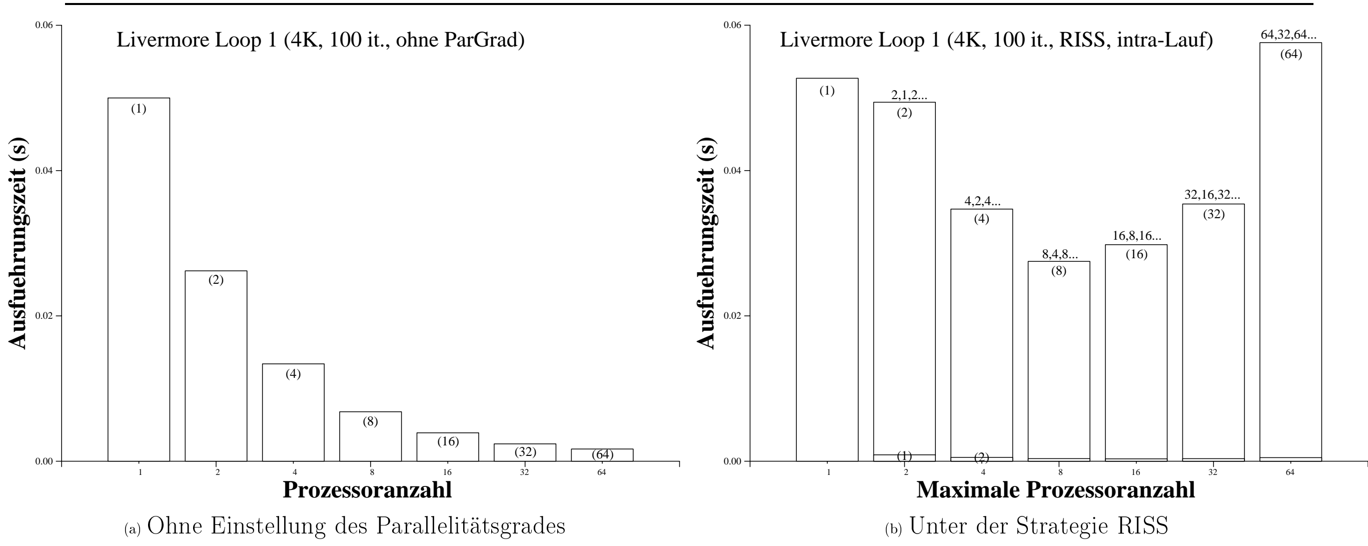
Livermore Loop 1						
Einstellung				Ziel	Optimum	Auswirkung
Ziel	Strategie	Art	Mehraufwand	erreicht?	gefunden?	
Ausführungszeit	RISS	intra-Lauf	< Faktor 35	nein	ja	+ Faktor 35
Ausführungszeit	RISS	inter-Lauf	< Faktor 35	nein	ja	+ Faktor 35
Effizienz	EISS	inter-Lauf	< 1%	ja	ja	+ Faktor 2,2

immer aufs Maximum für eine Problemgröße von 4K mit 100 Iterationen eingestellt hat (das Phänomen hat sich für Problemgrößen von 2 bzw. 32K wiederholt). Das ist bereits anhand der Abbildung 7.15a zu erwarten. Hier wird das Ziel der Einstellung nicht erreicht; die Ausführungszeit wird sogar bis zum Faktor 35 höher, wie der Abbildung 7.15b zu entnehmen ist. Erstens liegt es daran, daß der optimale Parallelitätsgrad in diesem Fall immer dem maximalen entspricht und zweitens, daß der Mehraufwand der Einstellung für dieses Programm stark ins Gewicht fällt, da die parallelisierbare Schleife zu feingranular ist. Aus diesen Gründen ist hier eine Inter-Lauf-Einstellung zu empfehlen. Sie wurde durchgeführt und konnte innerhalb von 2 Vorläufen des Programmes den richtigen optimalen Parallelitätsgrad finden.

Einstellung Effizienz: Dieser Einstellungsfall sollte dazu dienen, die Effizienz auf mindestens 50% einzustellen. Eine Inter-Lauf-Einstellung unter der Strategie EISS wurde vorgenommen, wodurch der optimale Parallelitätsgrad (16 für eine Problemgröße von 2K mit 100 Iterationen) gefunden wurde, wie in Abbildung 7.16 zu sehen ist. Die Effizienz konnte durch die Einstellung bis um Faktor 2,2 erhöht werden. Der Parallelitätsgrad für Problemgrößen von 4K sowie 32K wurde auch erfolgreich aufs Optimum eingestellt.

7.3.4.3 *Livermore Loop 3* (LL3)

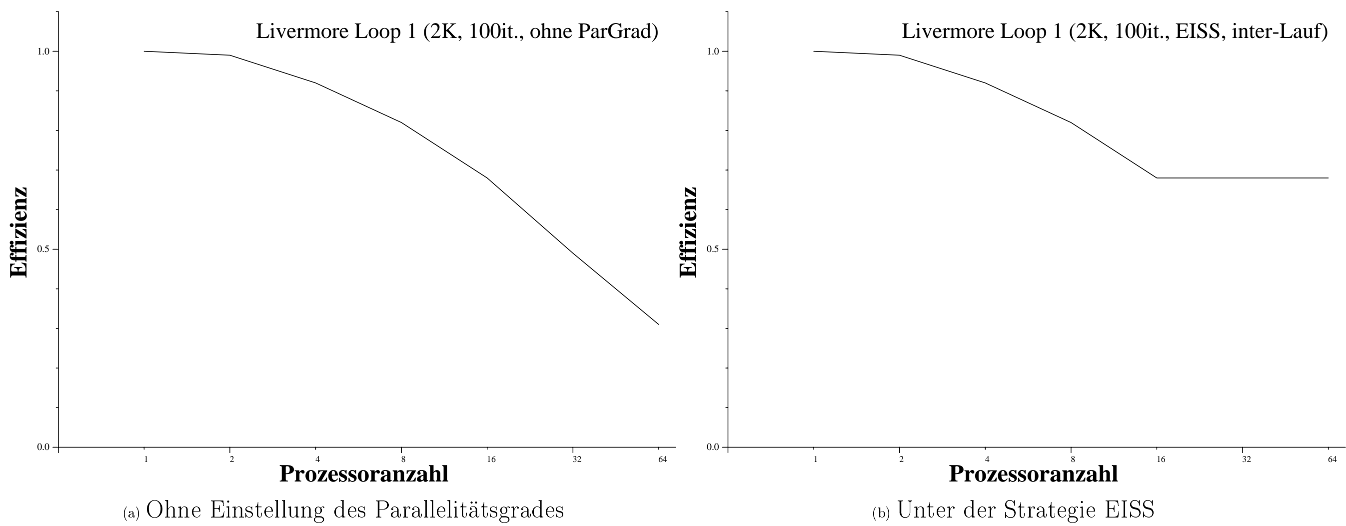
Tabelle 7.11 faßt die Ergebnisse für das Programm *Livermore Loop 3* zusammen. Jede betrachtete Einstellung hat eine einzige Einstellungsumgebung. Die Datenstrukturen, die wegen einer Änderung des Parallelitätsgrades umverteilt werden müssen, sind zwei eindimensionale Felder.



(a) Ohne Einstellung des Parallelitätsgrades

(b) Unter der Strategie RISS

Abbildung 7.15: LL1 unter der RISS-Strategie mit Intra-Lauf-Einstellung



(a) Ohne Einstellung des Parallelitätsgrades

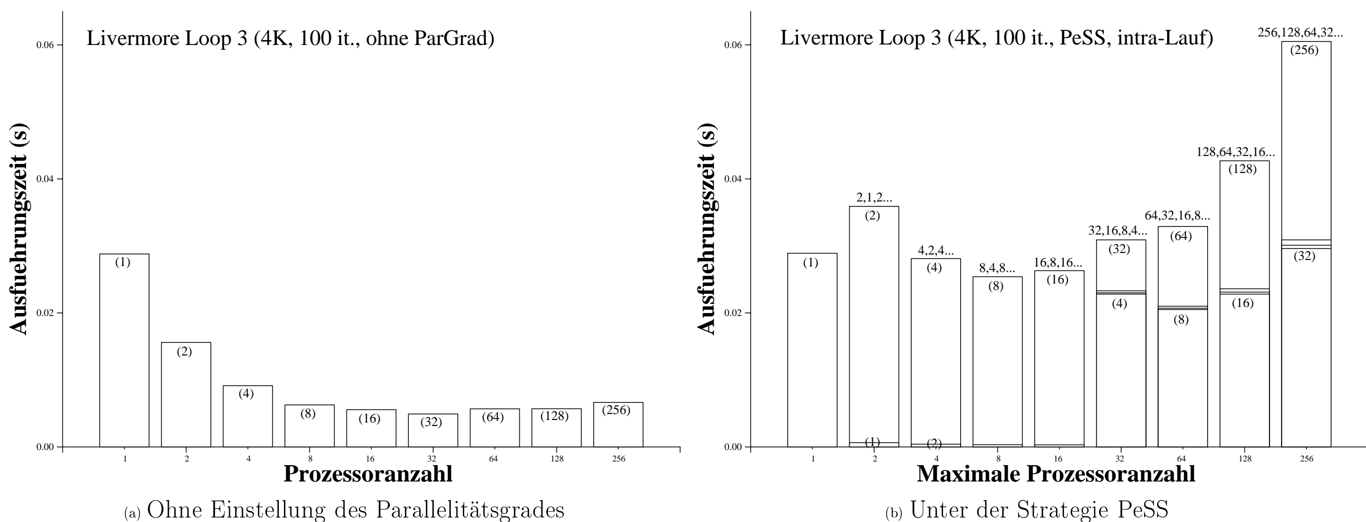
(b) Unter der Strategie EISS

Abbildung 7.16: Effizienz des LL1 unter der EISS-Strategie mit Inter-Lauf-Einstellung

Tabelle 7.11: Ergebnisse der Einstellung des Parallelitätsgrades für LL3

Livermore Loop 3						
Einstellung				Ziel	Optimum	Auswirkung
Ziel	Strategie	Art	Mehraufwand	erreicht?	gefunden?	
Ausführungszeit	PeSS	intra-Lauf	< Faktor 9	nein	ja	+ Faktor 9
Effizienz	EISS	inter-Lauf	< 1%	ja	ja	+ Faktor 36
Nutzen-Kosten	CBISS	inter-Lauf	< 1%	ja	ja	+ Faktor 3,2

Einstellung Ausführungszeit: Hier wurde die Strategie PeSS in einer Intra-Lauf-Einstellung zur Minimierung der Ausführungszeit des Programmes LL3 eingesetzt. Abbildung 7.17b zeigt, daß die Ausführungszeit dadurch eigentlich erhöht wurde (bis um Faktor 9). Dies ist auf die zu feine Granularität der parallelisierbaren Schleife zurückzuführen, welche den Mehraufwand der Einstellung nicht amortisieren kann. Der nach den Kriterien der PeSS-Strategie optimale Parallelitätsgrad konnte aber trotzdem gefunden werden. Dies geschah in drei Schritten, während die RISS-Strategie dafür fünf Schritte benötigt hätte. Ein Beispiel für die drei Schritte ist in Abbildung 7.17b für eine maximale Prozessoranzahl von 256 dargestellt. Dabei wurde der Parallelitätsgrad schrittweise auf 128, 64 und letztlich auf 32 eingestellt.

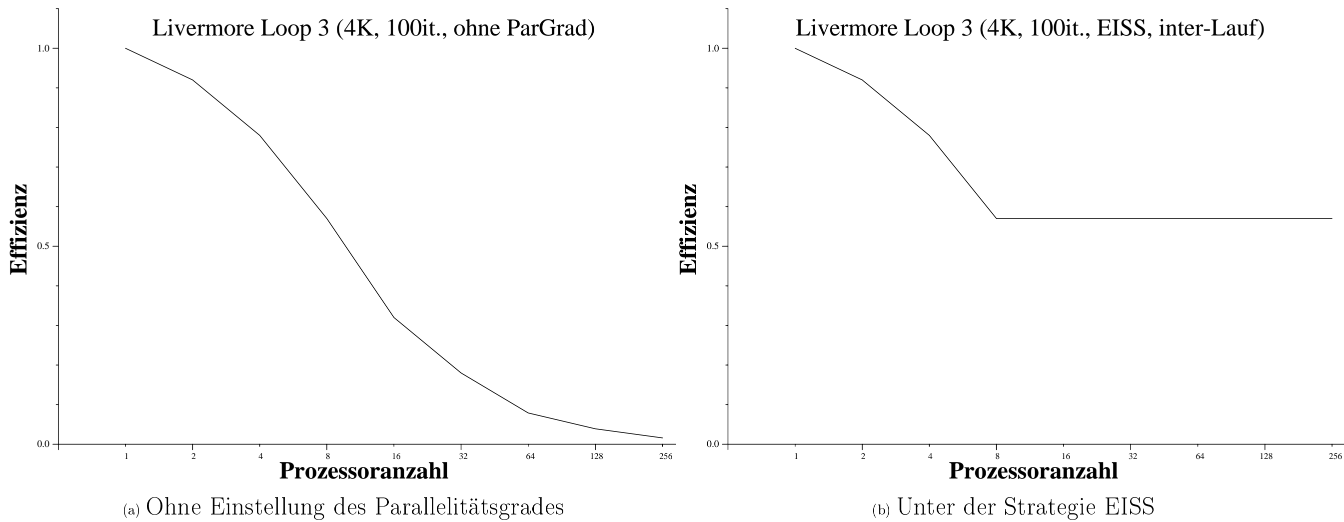


(a) Ohne Einstellung des Parallelitätsgrades

(b) Unter der Strategie PeSS

Abbildung 7.17: LL3 unter der PeSS-Strategie mit Intra-Lauf-Einstellung

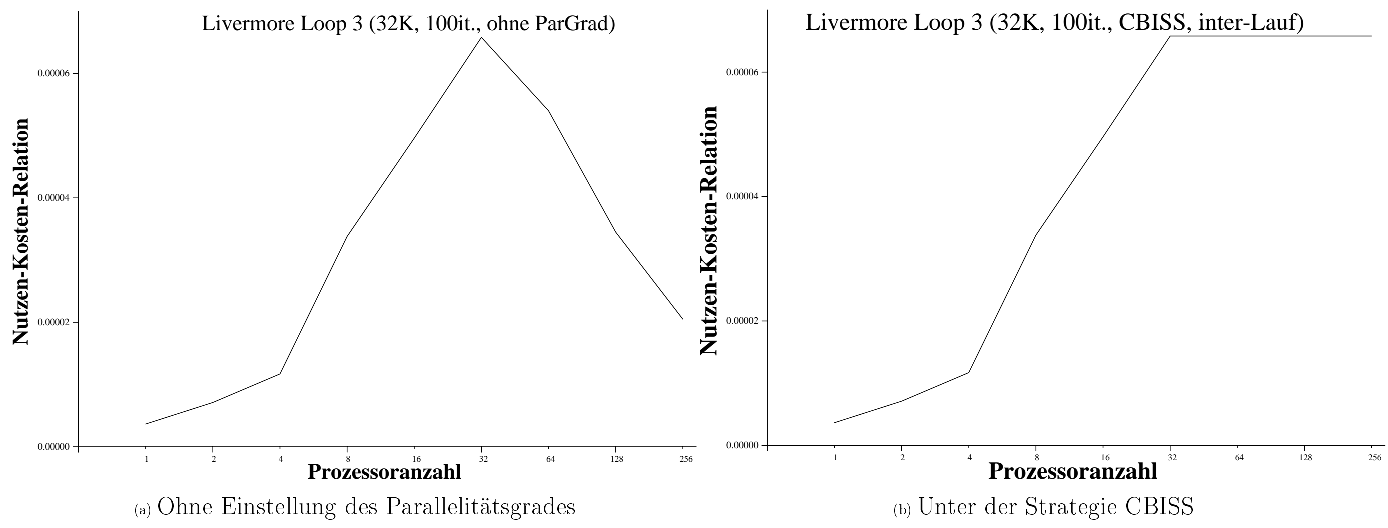
Einstellung Effizienz: Um die Effizienz des Programmes LL3 auf mindestens 50% zu halten, wurde eine Inter-Lauf-Einstellung des Parallelitätsgrades nach der EISS-Strategie vorgenommen. Bei einer Problemgröße von beispielsweise 4K mit 100 Iterationen wurde der Parallelitätsgrad richtig auf 8 eingestellt. Dadurch wurde eine maximale Erhöhung der Effizienz um Faktor 36 erreicht, wie aus einem Vergleich zwischen Abbildungen 7.18a und 7.18b herauszulesen ist.



(a) Ohne Einstellung des Parallelitätsgrades (b) Unter der Strategie EISS

Abbildung 7.18: Effizienz des LL3 unter der EISS-Strategie mit Inter-Lauf-Einstellung

Einstellung Nutzen-Kosten: Hier wird eine Inter-Lauf-Einstellung mit dem Ziel durchgeführt, die Nutzen-Kosten-Relation zu maximieren. Dieses Ziel wurde erreicht, wobei ein Beispiel dafür in Abbildung 7.19 gezeigt wird. Dabei wurde die Nutzen-Kosten-Relation für eine Problemgröße von 32K mit 100 Iterationen um Faktor 3,2 erhöht.



(a) Ohne Einstellung des Parallelitätsgrades (b) Unter der Strategie CBISS

Abbildung 7.19: Nutzen-Kosten-Relation des LL3 unter der CBISS-Strategie mit Inter-Lauf-Einstellung

7.3.4.4 *Livermore Loop 13* (LL13)

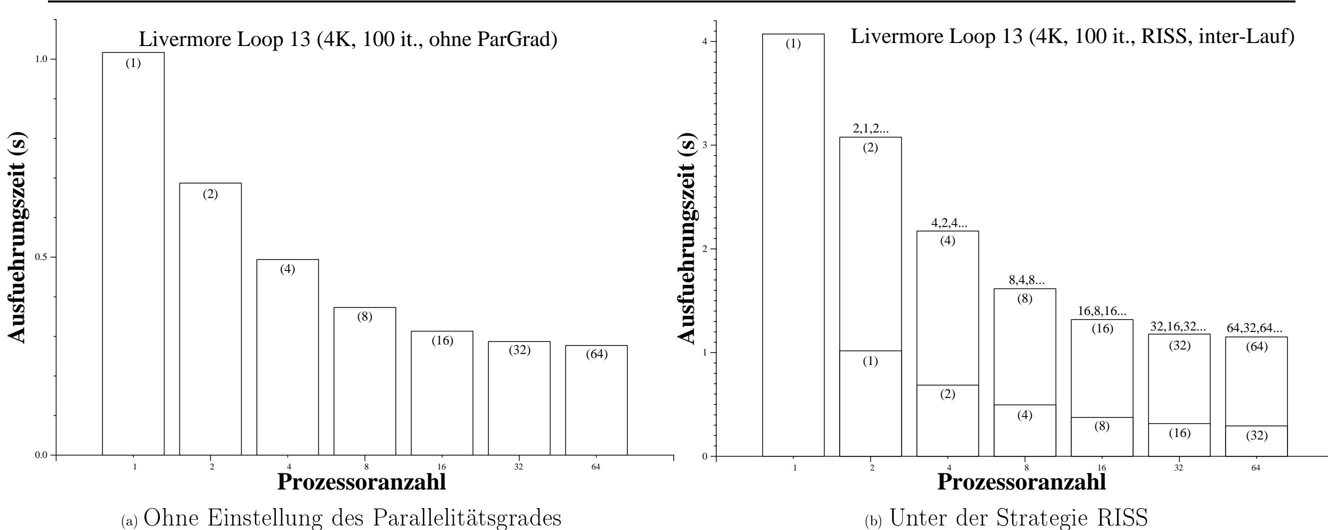
Hier werden die Ergebnisse für den *Livermore Loop 13* gezeigt. Tabelle 7.12 stellt eine Zusammenfassung der Ergebnisse dar. Jede betrachtete Einstellung hat eine einzige Einstellungsumgebung. Die umzuverteilende Datenstruktur ist ein eindimensionales Feld, das Eingabedaten enthält.

Tabelle 7.12: Ergebnisse der Einstellung des Parallelitätsgrades für LL13

Livermore Loop 13						
Einstellung				Ziel	Optimum	Auswirkung
Ziel	Strategie	Art	Mehraufwand	erreicht?	gefunden?	
Ausführungszeit	RISS	inter-Lauf	< 1%	ja	ja	○[a]
Effizienz	EISS	inter-Lauf	< 1%	ja	ja	+ Faktor 12
Nutzen-Kosten	CBISS	inter-Lauf	< 1%	ja	ja	+ Faktor 5

[a]Die minimale Ausführungszeit wurde bei dieser Einstellung immer mit der maximalen Prozessoranzahl erreicht, so daß die Ausführungszeit durch die Einstellung wegen niedriger noch höher geworden ist.

Einstellung Ausführungszeit: Bei dem ersten Einstellungsfall wurde die Ausführungszeit des Programmes LL13 unter der RISS-Strategie minimiert. Für eine Problemgröße von 4K mit 100 Iterationen hat der Parallelitätsgrad immer die maximale Prozessoranzahl durch eine Inter-Lauf-Einstellung erreicht, wie in Abbildung 7.20 zu sehen ist. Wenn die Problemgröße vervierfacht bzw. verachtfacht wurde, hat die Einstellung das gleiche Verhalten gezeigt.



(a) Ohne Einstellung des Parallelitätsgrades

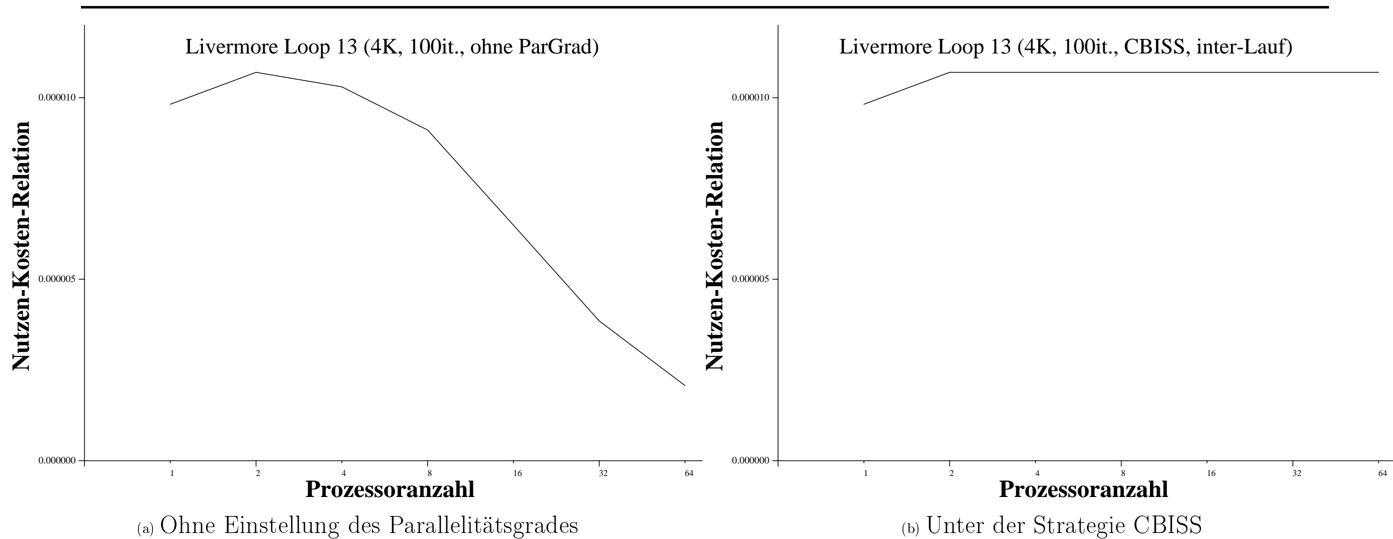
(b) Unter der Strategie RISS

Abbildung 7.20: LL13 unter der RISS-Strategie mit Inter-Lauf-Einstellung

Einstellung Effizienz: Hier werden vorgegebene Effizienzschranken mit Hilfe von der Strategie EISS in einer Inter-Lauf-Einstellung eingehalten. Eine Effizienzschranke von 50% wurde bei einem Parallelitätsgrad gleich 4 für eine Problemgröße von 4K mit 100 Iterationen erreicht. Bei einer Vervierfachung der Problemgröße wurde eine Effizienzschranke von 60% bei einem Parallelitätsgrad von 2 eingehalten. Dabei steigert die Effizienz bis zum Faktor 12 in Vergleich mit der Ausführung des Programmes ohne die Einstellung.

Einstellung Nutzen-Kosten: Die Strategie CBISS wird hier eingesetzt, um die Nutzen-Kosten-Relation durch eine Inter-Lauf-Einstellung zu maximieren. Dies wird für eine Problemgröße von 4K mit 100 Iterationen bei einem Parallelitätsgrad von 2 erreicht, wie in

Abbildung 7.21b dargestellt ist und anhand der Abbildung 7.21a zu erwarten war. Bei der Maximierung der Nutzen-Kosten-Relation wurde sie bis um Faktor 5 erhöht.



(a) Ohne Einstellung des Parallelitätsgrades

(b) Unter der Strategie CBISS

Abbildung 7.21: Nutzen-Kosten-Relation des LL13 unter der CBISS-Strategie mit Inter-Lauf-Einstellung

7.3.4.5 Veltran-Operator

Hier werden die für den Veltran-Operator erzielten Ergebnisse vorgestellt, wobei sie in der Tabelle 7.13 zusammengefaßt sind. Jede betrachtete Einstellung hat eine einzige Einstellungsumgebung, d.h. die Einstellung des Parallelitätsgrades findet jeweils nur an einer Stelle im Programm statt. Die Datenstrukturen, die wegen einer Umstellung des Parallelitätsgrades umverteilt werden müssen, sind zwei dreidimensionale Felder, die die Eingabedaten bzw. die partiellen Ergebnisse enthalten.

Für alle untersuchten Strategien blieben die Kosten für die Messungen und für die Datenumverteilung kleiner als 10% der Gesamtausführungszeit des Programmes.

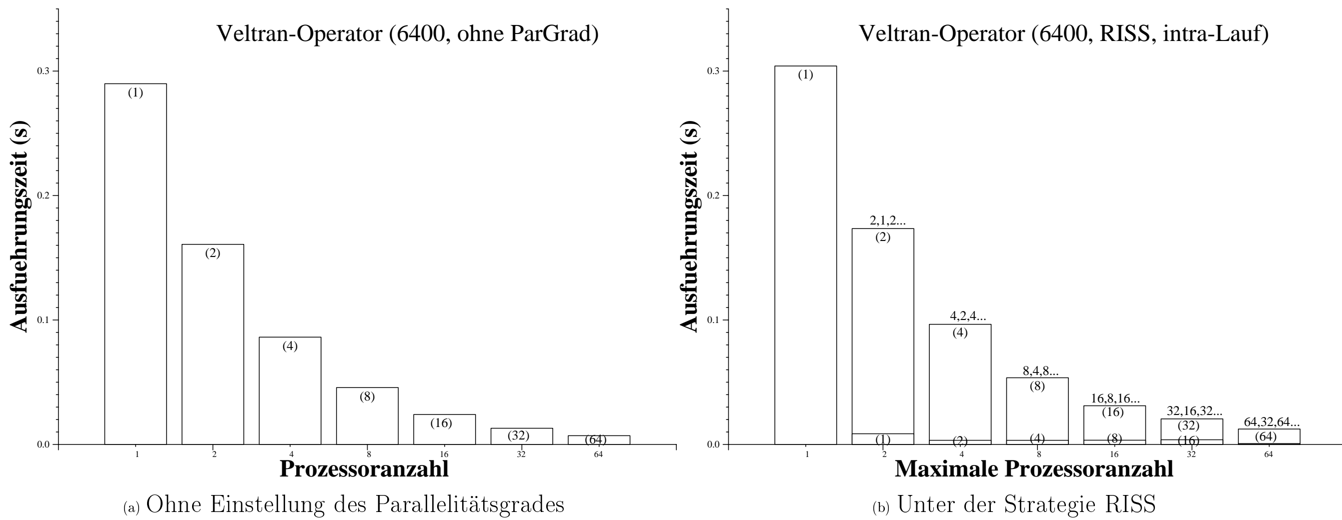
Tabelle 7.13: Ergebnisse der Einstellung des Parallelitätsgrades für den Veltran-Operator

Veltran-Operator						
Einstellung				Ziel	Optimum	Auswirkung
Ziel	Strategie	Art	Mehraufwand	erreicht?	gefunden?	
Ausführungszeit	RISS	intra-Lauf	< 10%	nein	ja	+ 75%
Effizienz	EISS	inter-Lauf	< 10%	ja	ja	+ 24%
Nutzen-Kosten	CBISS	inter-Lauf	< 10%	ja	ja	○[a]

[a] Die maximale Nutzen-Kosten-Relation wurde bei dieser Einstellung immer mit der maximalen Prozessoranzahl erreicht, so daß sie durch die Einstellung wegen niedriger noch höher geworden ist.

Einstellung Ausführungszeit: Die minimale Ausführungszeit des Veltran-Operators wird für alle untersuchten Fälle mit der maximalen Prozessoranzahl erreicht (siehe Abbildung

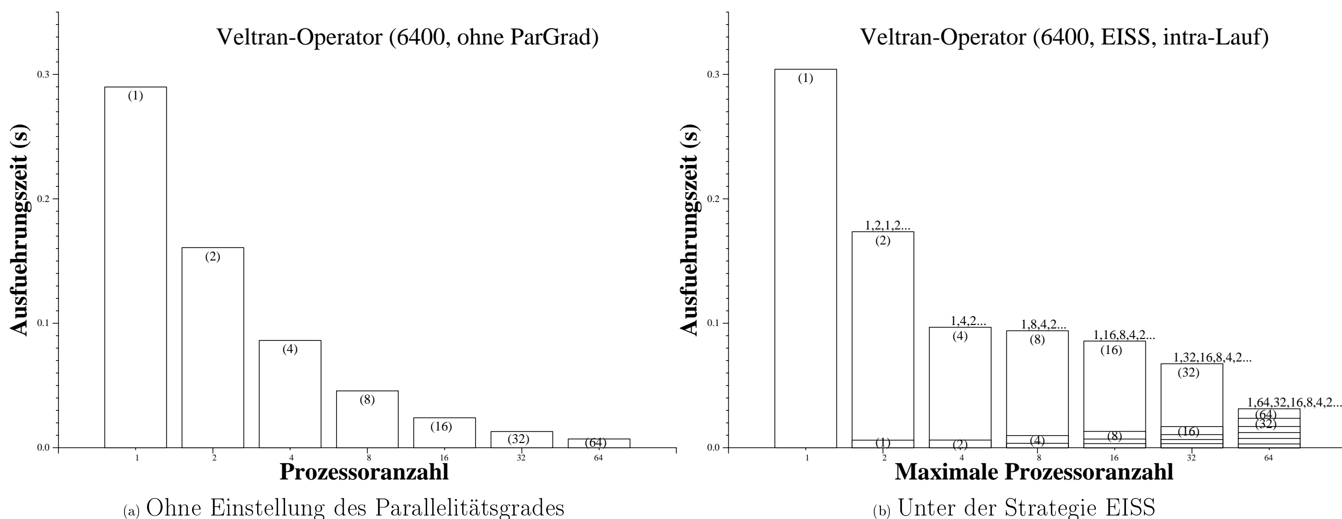
7.22a). Folge daraus ist, daß die Einstellung des Parallelitätsgrades über die Strategie RISS bzw. die Strategie PeSS (siehe Abschnitt 5.5.1) zu keiner Erniedrigung der Ausführungszeit des Programmes führen kann. Abbildung 7.22b zeigt, daß die Ausführungszeit des Programmes mit der Einstellung unter der Benutzung der Strategie RISS bis zu 75% höher wurde (Fall mit maximaler Prozessoranzahl gleich 64) als ohne Einstellung, da bei allen Fällen der Parallelitätsgrad erwartungsgemäß auf das Maximum eingestellt wird. D.h. die Einstellung bringt keinen Gewinn, sonder zusätzliche Kosten. Im Gegensatz zu der Ausführungszeit können aber die Effizienz und die Nutzen-Kosten-Relation doch verbessert werden. Der Parallelitätsgrad wurde hier mit dem Ziel eingestellt, die Effizienz des Programmes zu erhöhen, und dadurch die Auslastung des Gesamtsystems zu verbessern. Daher wurde die Strategie EISS (siehe Abschnitt 5.5.1) angewendet.



(a) Ohne Einstellung des Parallelitätsgrades

(b) Unter der Strategie RISS

Abbildung 7.22: Veltran-Operator unter der RISS-Strategie mit Intra-Lauf-Einstellung

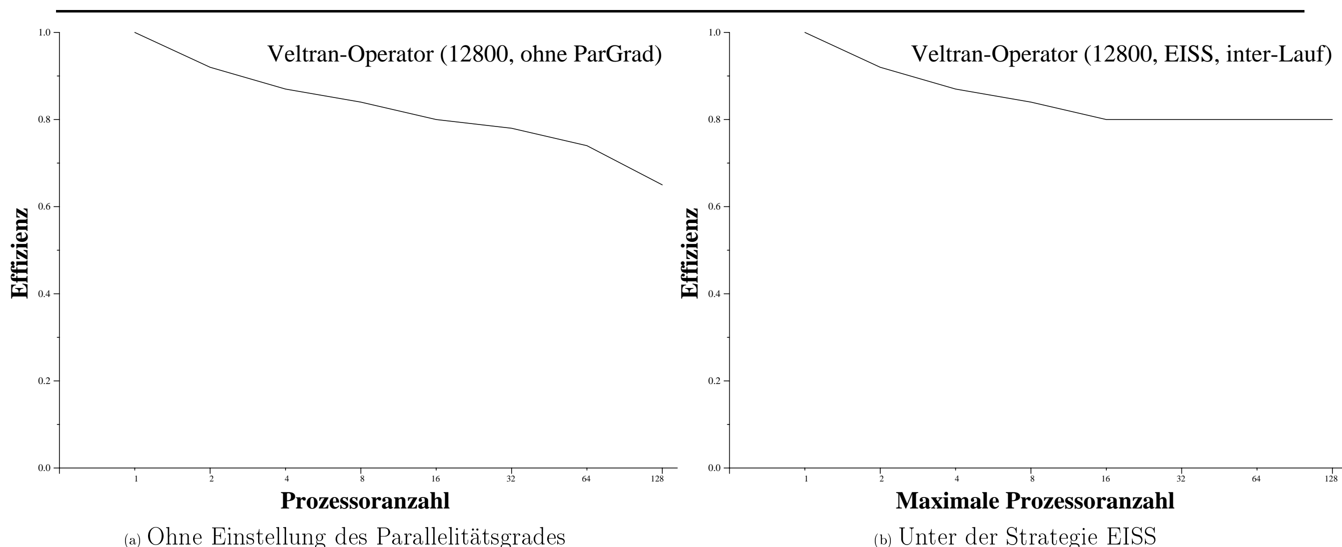


(a) Ohne Einstellung des Parallelitätsgrades

(b) Unter der Strategie EISS

Abbildung 7.23: Veltran-Operator unter der EISS-Strategie mit Intra-Lauf-Einstellung

Einstellung Effizienz: Im Vergleich zu einer Ausführung des Programmes ohne Einstellung des Parallelitätsgrades lief das Programm z.B. mit 16 Prozessoren 4 mal langsamer unter der Benutzung der EISS-Strategie, wie Abbildungen 7.23a und b zeigen. Für Problemgrößen von 6400 bzw. 12800 Elementen wurde der Parallelitätsgrad richtig auf 2 unter der Strategie EISS eingestellt, wenn die Effizienzschranke 90% betrug. Dabei ist die Effizienz um 12% höher geworden als bei der Ausführung ohne Einstellung des Parallelitätsgrades.



(a) Ohne Einstellung des Parallelitätsgrades

(b) Unter der Strategie EISS

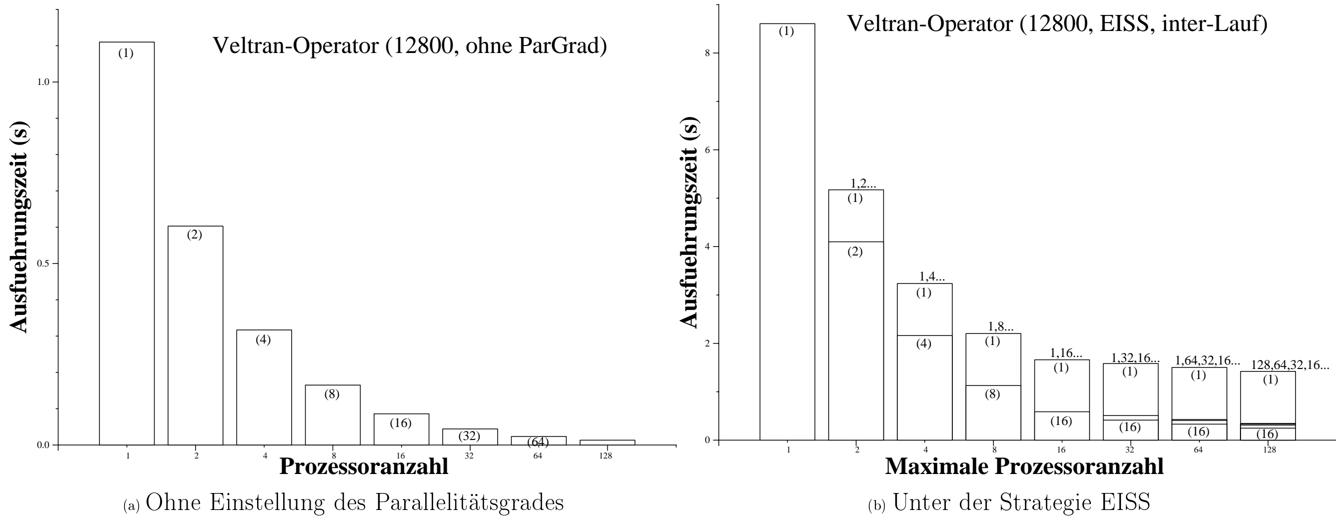
Abbildung 7.24: Effizienz des Veltran-Operators unter der EISS-Strategie mit Inter-Lauf-Einstellung

Für eine Problemgröße gleich 12800 Elementen und eine Effizienzschranke von 80% wurde der Parallelitätsgrad unter der Strategie EISS als Inter-Lauf richtig auf 16 eingestellt. Die Effizienz des Programmes ohne Einstellung des Parallelitätsgrades verhält sich wie in Abbildung 7.24a gezeigt. Anzumerken ist, daß die Effizienz für Parallelitätsgrade, die von 1 bis 16 variieren, mindestens 80% ist, was die Korrektheit der Einstellung unter der Strategie EISS beweist. Bei z.B. einer maximalen Prozessoranzahl gleich 128 läuft das Programm nach der Einstellung auf 16 Prozessoren um Faktor 6 langsamer (siehe Abbildung 7.25), aber dafür wird die Effizienz um ca. 24% höher als ohne Einstellung (ohne Einstellung geht man hier also von 128 Prozessoren aus). Abbildung 7.25 stellt die Inter-Lauf-Einstellung des Parallelitätsgrades unter der Strategie EISS für 8 Iterationen dar.

Einstellung Nutzen-Kosten: Der Parallelitätsgrad wurde hier mit dem Ziel eingestellt, dem Programm nur so viele Prozessoren zuzuordnen, daß die durch den Einsatz mehrerer Prozessoren verursachten Kosten durch eine Reduzierung der Ausführungszeit wiedergewonnen werden. Dafür wurde die Strategie CBISS (siehe Abschnitt 5.5.1) angewendet und die Nutzen-Kosten-Relation maximiert.

Abbildung 7.26 zeigt die Entwicklung der Nutzen-Kosten-Relation für den Veltran-Operator bei steigender Prozessoranzahl, wobei die Problemgröße gleich 12800 ist. Die maximale Nutzen-Kosten-Relation wird bei einem Parallelitätsgrad von 128 erreicht.

Abbildung 7.27 zeigt eine Inter-Lauf-Einstellung des Parallelitätsgrades des Veltran-Operators nach der Strategie CBISS für eine Problemgröße gleich 12800. Der Parallelitätsgrad wurde bis zu 128 Prozessoren immer richtig auf die maximale Prozessoranzahl ein-



(a) Ohne Einstellung des Parallelitätsgrades

(b) Unter der Strategie EISS

Abbildung 7.25: Veltran-Operator unter der EISS-Strategie mit Inter-Lauf-Einstellung

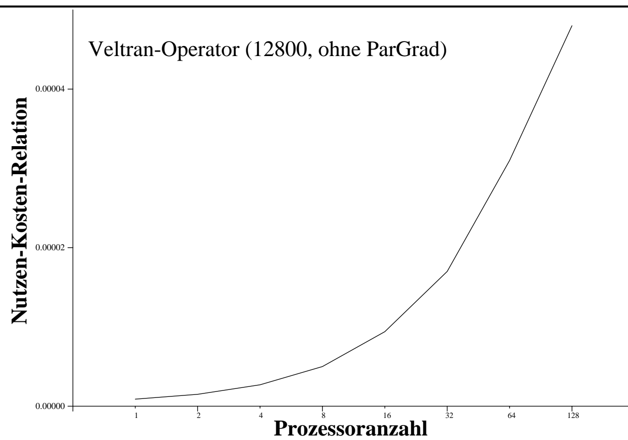
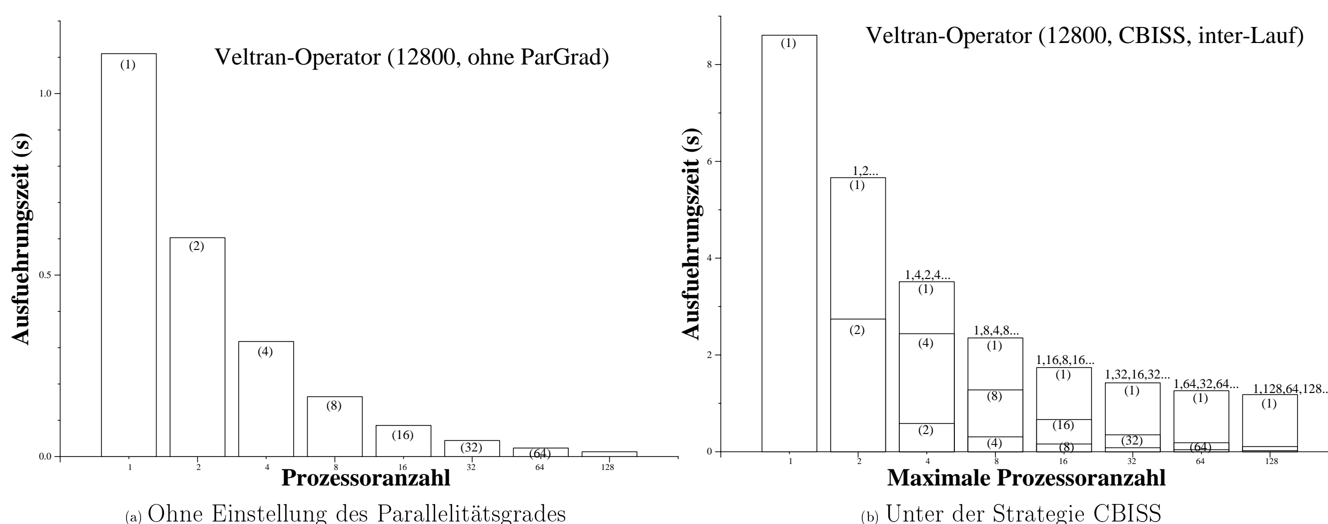


Abbildung 7.26: Entwicklung der Nutzen-Kosten-Relation für den Veltran-Operator

gestellt, wie es anhand der Kurve der Nutzen-Kosten-Relation in Abbildung 7.26 zu erwarten war.



(a) Ohne Einstellung des Parallelitätsgrades

(b) Unter der Strategie CBISS

Abbildung 7.27: Veltran-Operator unter der CBISS-Strategie mit Inter-Lauf-Einstellung

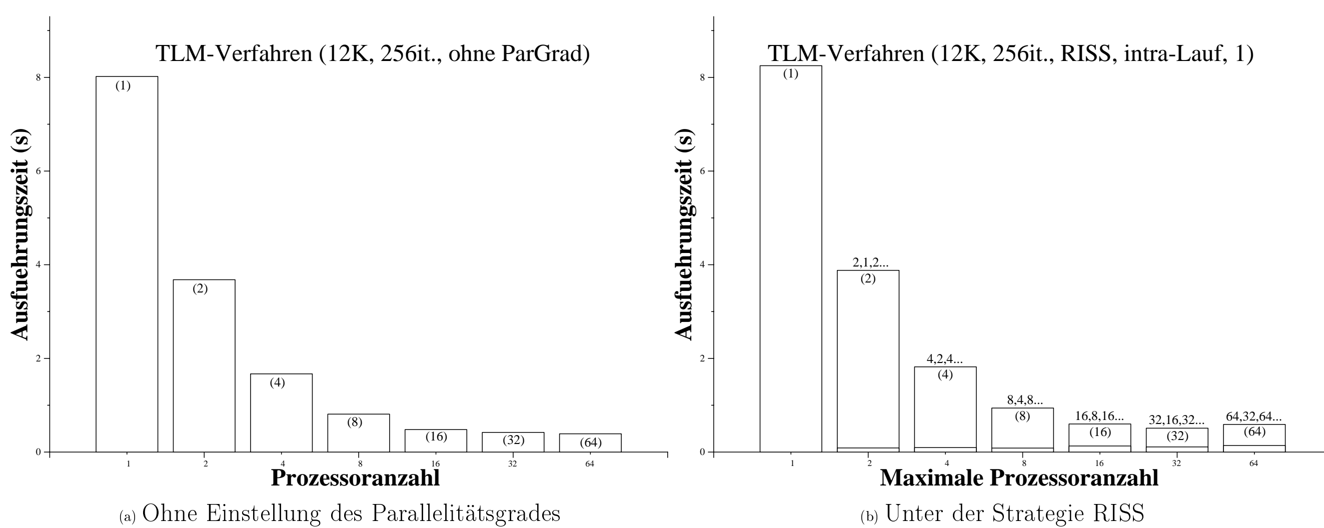
7.3.4.6 TLM-Verfahren

Tabelle 7.14 stellt die erzielten Ergebnisse für das TLM-Verfahrendar. Für diese Anwendung wurden drei parallelisierbare Schleifen auf die Einstellung des Parallelitätsgrades hin untersucht. Zudem wurde ein Parallelisierungsfall betrachtet, bei dem mehrere Einstellungsumgebungen vorhanden waren. Die bei einer Änderung in Parallelitätsgrad umzuverteilenden Datenstrukturen sind in den untersuchten Fällen unterschiedlich und werden mit der entsprechenden Diskussion der Fälle präsentiert. Bei allen präsentierten Fällen handelt es sich um Intra-Lauf-Einstellungen. Für das TLM-Verfahren wurde keine Inter-Lauf-Einstellung durchgeführt, weil dies nur mit aufwendigen Änderungen in der Dateneingabe dieser Anwendung gebunden gewesen wäre. Weil einer der Randbedingungen der vorliegenden Arbeit die Vornahme möglichst wenig Änderungen im parallelen Programm vorsieht, wird hier keine Inter-Lauf-Einstellung betrachtet.

Tabelle 7.14: Ergebnisse der Einstellung des Parallelitätsgrades für das TLM-Verfahren

TLM-Verfahren						
Einstellung				Ziel	Optimum	Auswirkung
Ziel	Strategie	Art	Mehraufwand	erreicht?	gefunden?	
Nutzen-Kosten	CBISS	intra-Lauf	< 30%	nein	ja	+ 23%
Nutzen-Kosten	CBISS	intra-Lauf	< Faktor 3	nein	ja	+ Faktor 2
Nutzen-Kosten	CBISS	intra-Lauf	< Faktor 1,5	nein	ja	+ 80%
Nutzen-Kosten	CBISS	intra-Lauf	dito	nein	ja	dito

Einstellung Nutzen-Kosten Schleife 1: Bei der Benutzung der Strategie RISS in der Schleife 1 wurde der Parallelitätsgrad immer auf das Maximum eingestellt, wie es auf Basis der Meßergebnisse im Fall ohne Einstellung (siehe Abbildung 7.28a) zu erwarten wäre. Die in dieser Schleife umzuverteilenden Strukturen sind *Knoten*, *Randknoten*, *Anregung* und *Empfänger* (siehe Abschnitt 7.1.6).

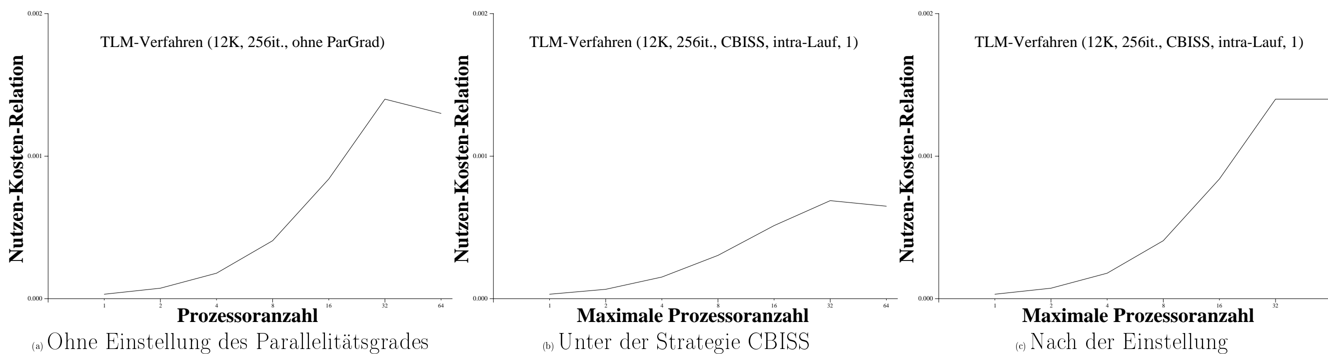


(a) Ohne Einstellung des Parallelitätsgrades

(b) Unter der Strategie RISS

Abbildung 7.28: TLM-Verfahren unter der Strategie RISS für die Schleife 1

Anhand der Abbildung 7.28b kann festgestelt werden, daß die Ausführungszeit für niedrige maximale Prozessoranzahlen aufgrund der Einstellung des Parallelitätsgrades unter der Strategie RISS geringfügig (ca. 10%) höher geworden ist; für höhere maximale Prozessoranzahlen hat sie bis zu 50% zugenommen. Aus diesem Grund wurde die Strategie RISS für das TLM-Verfahren nicht mehr angewendet, sondern die Strategie CBISS (siehe Abschnitt 5.5.1). Das dadurch erzielte Ergebnis wird in Abbildung 7.29 gezeigt.



(a) Ohne Einstellung des Parallelitätsgrades

(b) Unter der Strategie CBISS

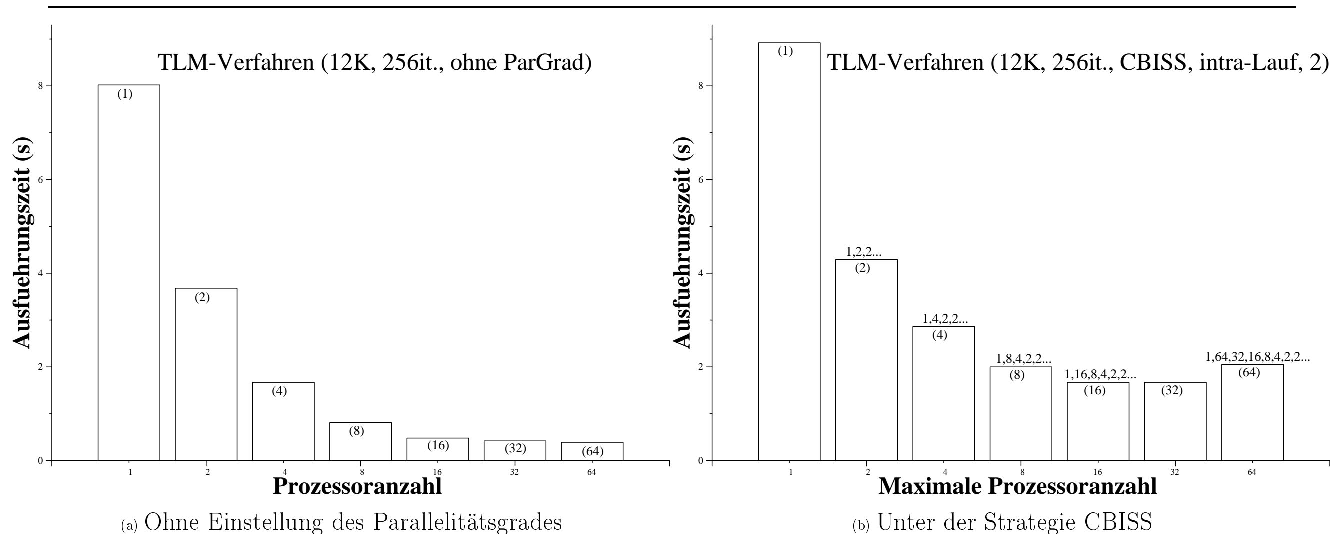
(c) Nach der Einstellung

Abbildung 7.29: Nutzen-Kosten-Relation des TLM-Verfahrens unter der Strategie CBISS für die Schleife 1

Bis auf eine maximale Prozessoranzahl gleich 32 hat die Strategie CBISS den Parallelitätsgrad auf die maximale Prozessoranzahl eingestellt. Als 64 Prozessoren zur Verfügung standen, wurde der Parallelitätsgrad auf 32 eingestellt, wie es anhand der Kurve in Abbildung 7.29a zu erwarten war. Weil die Datenumverteilung für diese Anwendung mit hohen Kosten gebunden ist, ist die Nutzen-Kosten-Relation während der Einstellung (siehe Abbildung

7.29b) immer niedriger als ohne Einstellung geworden. Durch die Einstellung des Parallelitätsgrades konnte der optimale Parallelitätsgrad aber trotzdem gefunden werden. Wenn das Optimum schon bekannt ist, wird die Nutzen-Kosten-Relation immer direkt darauf eingestellt, wie Abbildung 7.29c zeigt. Dadurch wird eine Erhöhung der Nutzen-Kosten-Relation um ca. 23% erzielt, wobei sich die Ausführungszeit um ca. 27% erhöht. Obwohl eine Inter-Lauf-Einstellung für diesen Fall zu empfehlen wäre, wurde diese Art der Einstellung aus den o.g. Aufwandsgründen nicht vorgenommen.

Einstellung Nutzen-Kosten Schleife 2: Der Parallelitätsgrad wurde hier mit dem Ziel eingestellt, die Nutzen-Kosten-Relation der Schleife 2 zu maximieren. Daher wurde die Strategie CBISS eingesetzt. Die in dieser Schleife umzuverteilenden Datenstrukturen sind *Knoten* und *Randknoten* (siehe Abschnitt 7.1.6).



(a) Ohne Einstellung des Parallelitätsgrades (b) Unter der Strategie CBISS

Abbildung 7.30: TLM-Verfahren unter der Strategie CBISS für Schleife 2

Die Nutzen-Kosten-Relation hat ihr Maximum bei einer Prozessoranzahl gleich 2 erreicht. Die Ausführungszeit des Programmes war bis zum Faktor 4 höher, wie Abbildung 7.30 zeigt; dafür ist die Nutzen-Kosten-Relation um Faktor 2 gestiegen. Wenn der optimale Parallelitästgrad für die betroffene Schleife schon bekannt war, d.h. wenn er in der ParGrad-Datenbank (siehe Abschnitt 5.9) vorhanden war, konnte die Zunahme der Ausführungszeit um ca. 20% reduziert werden. Dabei wurde der optimale Parallelitästgrad aus der Datenbank herausgelesen und der Parallelitästgrad bereits zu Anfang der betroffenen Schleife auf das Optimum eingestellt.

Die Säulen-Segmente können in Abbildung 7.30b nicht gesehen werden, weil die Ausführungszeit der Schleife, in der die Einstellung des Parallelitätsgrades stattfindet, sehr gering ist im Vergleich zu der Ausführungszeit des gesamten Programmes. Wie in Abschnitt 7.3.3 erläutert, stellt eine Säule die Ausführungszeit des gesamten Programmes dar.

Einstellung Nutzen-Kosten Schleife 3: Der Parallelitätsgrad wurde hier mit dem Ziel eingestellt, die Nutzen-Kosten-Relation der Schleife 3 zu maximieren. Dafür wurde die Strategie CBISS angewendet. Hier muß die Datenstruktur *Knoten* (siehe 7.1.6) bei einer Änderung des Parallelitätsgrades umverteilt werden.

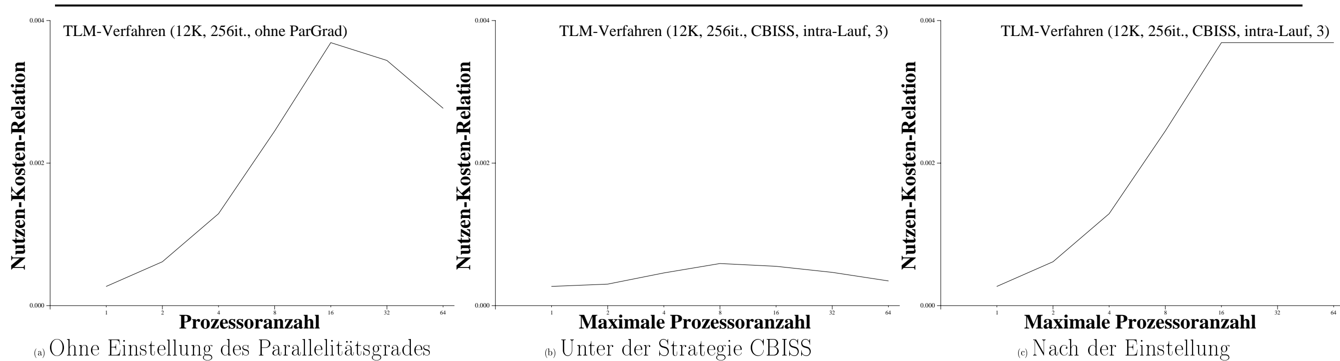


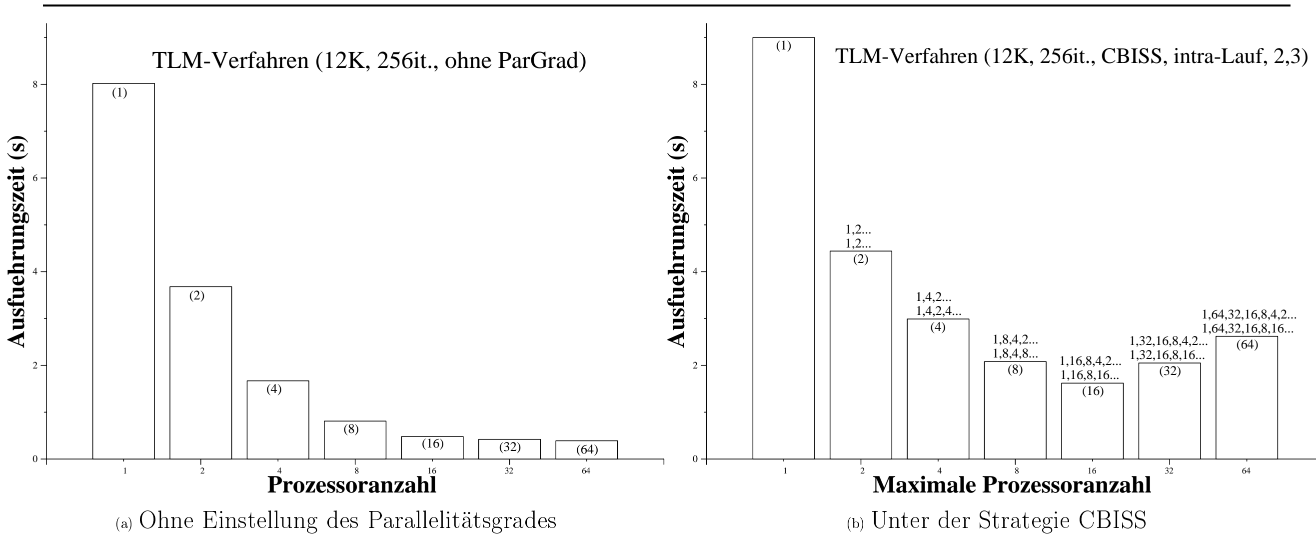
Abbildung 7.31: Nutzen-Kosten-Relation des TLM-Verfahrens unter der Strategie CBISS für Schleife 3

Die Nutzen-Kosten-Relation erreicht ihr Maximum bei einer Prozessoranzahl gleich 16 für die dritte untersuchte Schleife, wie in Abbildung 7.31a zu sehen ist. Weil die Datenumverteilung für diese Anwendung mit hohen Kosten gebunden ist, ist die Nutzen-Kosten-Relation während der Einstellung (siehe Abbildung 7.31b) stets niedriger als ohne die Einstellung. Durch die Einstellung des Parallelitätsgrades konnte der optimale Parallelitätsgrad aber trotzdem gefunden werden. Wenn das Optimum bereits bekannt ist, wird die Nutzen-Kosten-Relation vorab darauf eingestellt, wie Abbildung 7.31c zeigt. Die Nutzen-Kosten-Relation ist bis zu 80% gestiegen, wobei die Ausführungszeit des Programmes bis zum Faktor 2 höher geworden ist. Obwohl eine Inter-Lauf-Einstellung hier eher zu empfehlen wäre, wurde diese Art der Einstellung aus den o.g. Aufwandsgründen nicht untersucht.

Einstellung Nutzen-Kosten Schleifen 2 und 3: Für das TLM-Verfahren wurden noch die Parallelitätsgrade der Schleifen 2 und 3 im selben Lauf unter der Strategie CBISS eingestellt. Es wurden zwei Einstellungsumgebungen spezifiziert, wobei die Menge der umzuverteilenden Datenstrukturen in den zwei Fällen teilweise unterschiedlich sind. Wie auf Basis der Ergebnisse der Einzelschleifen (Abbildungen 7.30 und 7.31) bereits zu erwarten ist, wurden die Parallelitätsgrade auf 2 bzw. 16 eingestellt. Schleife 2 sowie Schleife 3 sind innere, aufeinanderfolgende Schleifen einer Verschachtelung. Die Parallelitätsgrade werden nur in der ersten Iteration der äußeren Schleife (Schleife 1) eingestellt. An jeden der darauffolgenden Iterationen der Schleife 1 wird der Parallelitätsgrad auf den zu dem Zeitpunkt bereits bekannten optimalen Wert sowohl für Schleife 2 als auch für Schleife 3 eingestellt, sofern er von der maximalen Prozessoranzahl überhaupt abweicht. Am Ende der Schleifen 2 und 3 wird der Parallelitätsgrad wieder auf den Wert, den er vor dem Beginn der Einstellung hatte, zurückgesetzt und die restlichen Berechnungen in der Schleife 1 ausgeführt. Abbildung 7.32 stellt die Ausführungszeit in Abhängigkeit von der maximalen Prozessoranzahl für die Einstellung des Parallelitätsgrades mit den zwei Einstellungsumgebungen dar.

7.3.4.7 PDE-Löser

Hier werden die für den PDE-Löser erzielten Ergebnisse präsentiert, wobei sie in der Tabelle 7.15 zusammengefaßt sind. Jede berichtete Einstellung hat eine einzige Einstellungsumgebung, d.h. die Einstellung des Parallelitätsgrades findet nur an einer Stelle im Programm statt. Die Datenstrukturen, die wegen einer Änderung im Parallelitätsgrad umverteilt werden müssen, sind für den PDE-Löser vier Felder, die die Eingabedaten bzw. partielle Ergeb-



(a) Ohne Einstellung des Parallelitätsgrades (b) Unter der Strategie CBISS

Abbildung 7.32: TLM-Verfahren unter der Strategie CBISS für Schleife 2 und 3

nisse enthalten.

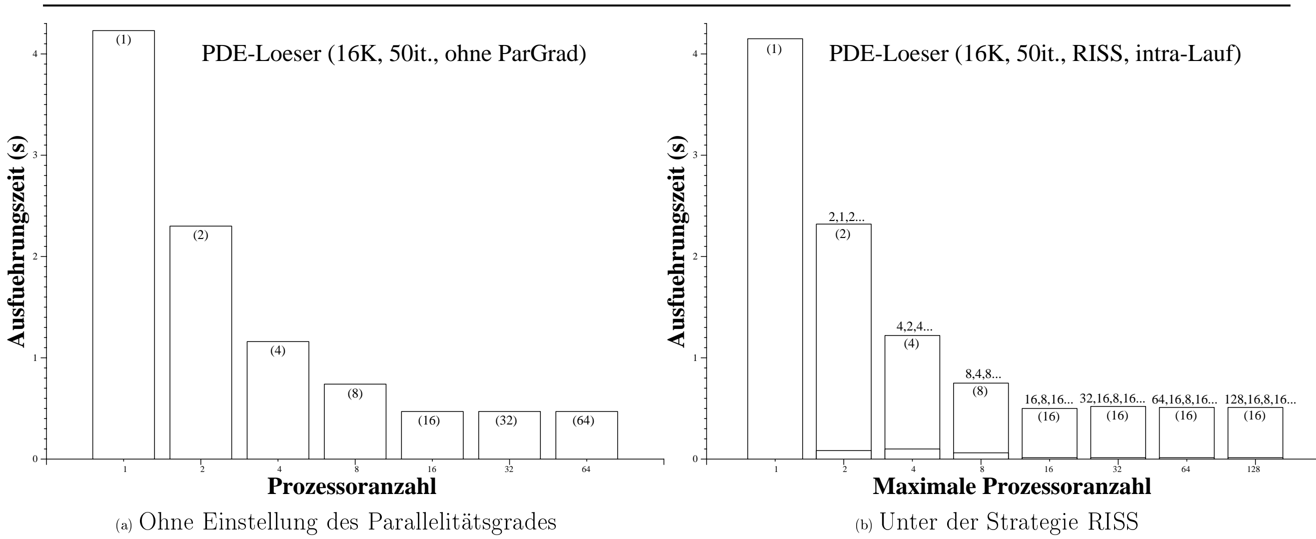
Tabelle 7.15: Ergebnisse der Einstellung des Parallelitätsgrades für den PDE-Löser

PDE-Löser						
Einstellung				Ziel	Optimum	Auswirkung
Ziel	Strategie	Art	Mehraufwand	erreicht?	gefunden?	
Ausführungszeit	RISS	inter-Lauf	< 15%	ja	ja	− 10%
Effizienz	EISS	intra-Lauf	< 50%	ja	ja	+ Faktor 7
Nutzen-Kosten	CBISS	intra, inter	< 45%	ja	ja	+ Faktor 8

Einstellung Ausführungszeit: Der Parallelitätsgrad wurde hier mit dem Ziel eingestellt, die Ausführungszeit zu minimieren. Dazu wurde die Strategie RISS (siehe Abschnitt 5.5.1) eingesetzt.

Abbildung 7.33a zeigt das Verhalten der Ausführungszeit des Programmes bei einer Problemgröße gleich 16K, 50 Iterationen und einer zunehmenden Prozessoranzahl, wenn der Parallelitätsgrad nicht eingestellt wird. Bei einer Einstellung unter der Benutzung der Strategie RISS wird der Parallelitätsgrad richtig auf 16 eigestellt, wie in Abbildung 7.33b zu sehen ist. Obwohl der Parallelitätsgrad z.B. von 64 auf 16 reduziert wurde (d.h. daß die minimale Ausführungszeit bei einer Prozessoranzahl gleich 16 erreicht wurde, wie Abbildung 7.33a bestätigt), hat sich die Gesamtausführungszeit um ca. 8% verschlechtert. Dieser Effekt ist auf die Datenumverteilung zurückzuführen. Bei einer Verdoppelung der Problemgröße erhöht sich die Gesamtausführungszeit um ca. 12%.

Da die Intra-Lauf-Einstellung des Parallelitätsgrades unter der Strategie RISS wegen der für diese Anwedung aufwendigen Datenumverteilung zu keiner Reduzierung der Ausführungszeit führt, wurde eine Inter-Lauf-Einstellung über sechs Iterationen unter derselben Strategie vorgenommen. Das Ergebniss dieser Einstellung ist in Abbildung 7.34 zu sehen. Der Parallelitätsgrad wurde dabei wie erwartet auf 16 eingestellt. Nach diesem Vorlauf kann

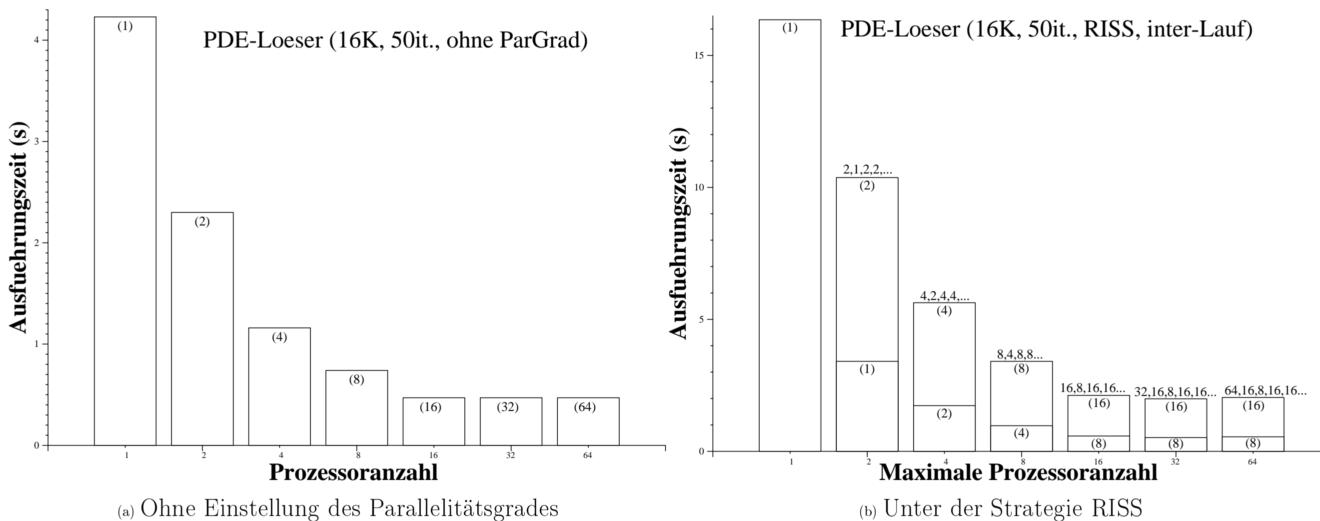


(a) Ohne Einstellung des Parallelitätsgrades

(b) Unter der Strategie RISS

Abbildung 7.33: PDE-Löser bei einer Intra-Lauf-Einstellung unter der Strategie RISS

der Benutzer sein Programm für die betrachtete Problemgröße immer mit 16 Prozessoren starten, wodurch die Ausführungszeit seinen minimalen Wert erreicht.



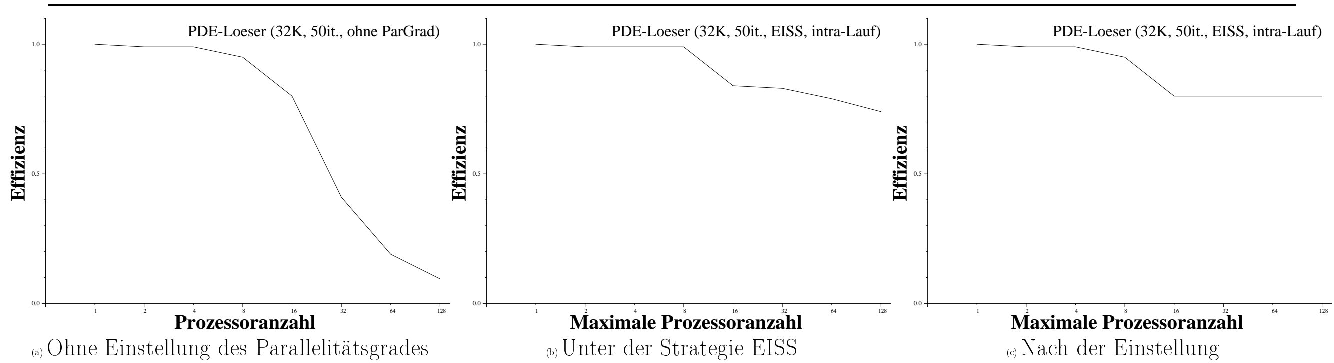
(a) Ohne Einstellung des Parallelitätsgrades

(b) Unter der Strategie RISS

Abbildung 7.34: PDE-Löser bei einer Inter-Lauf-Einstellung unter der Strategie RISS

Einstellung Effizienz: Der Parallelitätsgrad wurde hier mit dem Ziel eingestellt, die Effizienz des Programmes so zu steigern, daß sie sich nicht unterhalb einer angegebenen Schranke befindet. Das Endziel war, die Auslastung des Gesamtsystems dadurch zu verbessern. Die Strategie EISS (siehe Abschnitt 5.5.1) kam dabei zum Einsatz.

Abbildung 7.35 zeigt das Verhalten der Effizienz für den PDE-Löser bei einer Problemgröße gleich 32K, 50 Iterationen und einer Effizienzschranke von 70%. Der Parallelitätsgrad wurde dabei richtig auf 8 eingestellt, wobei die Effizienz bereits während der Einstellung des Parallelitätsgrades bis zum Faktor 7,5 gestiegen ist (siehe Abbildung 7.35b) und die Ausführungszeit bis zum Faktor 2 erhöht wurde. Wenn der optimale Parallelitätsgrad vorab

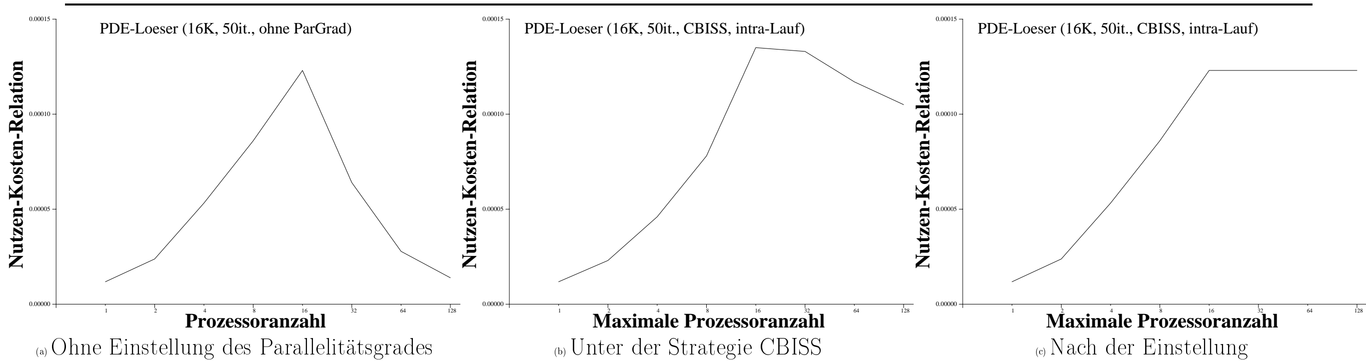


(a) Ohne Einstellung des Parallelitätsgrades (b) Unter der Strategie EISS (c) Nach der Einstellung

Abbildung 7.35: Entwicklung der Effizienz des PDE-Lösers unter der Strategie EISS mit Intra-Lauf-Einstellung

bekannt ist und eine entsprechende Einstellung zu Anfang vorgenommen wird, verhält sich die Effizienz wie in Abbildung 7.35c präsentiert. Bei einer Problemgröße gleich 16K, 50 Iterationen wurde der Parallelitätsgrad auf 16 nach der Strategie EISS richtig eingestellt, wenn die Effizienzschranke 50% betrug. Dabei wurde die Effizienz um bis zum Faktor 8 erhöht und die Ausführungszeit bis zu 25% gesteigert.

Einstellung Nutzen-Kosten: Die Einstellung des Parallelitätsgrades verfolgt hier das Ziel der Maximierung der Nutzen-Kosten-Relation basierend auf der Strategie CBISS (siehe Abschnitt 5.5.1). Die Entwicklung der Nutzen-Kosten-Relation ohne Einstellung des Parallelitätsgrades ist in Abbildung 7.36a gezeigt. Bereits während der Einstellung war die Nutzen-Kosten-Relation bis zum Faktor 7,5 höher als ohne Einstellung, wie in Abbildung 7.36b zu sehen ist. Nach der Einstellung des Parallelitätsgrades, d.h. wenn das Optimum bekannt ist und aus der Parallelitätsgrad-Datenbank (siehe Abschnitt 5.9) herausgelesen werden kann, verhält sich die Nutzen-Kosten-Relation wie in Abbildung 7.36c dargestellt.

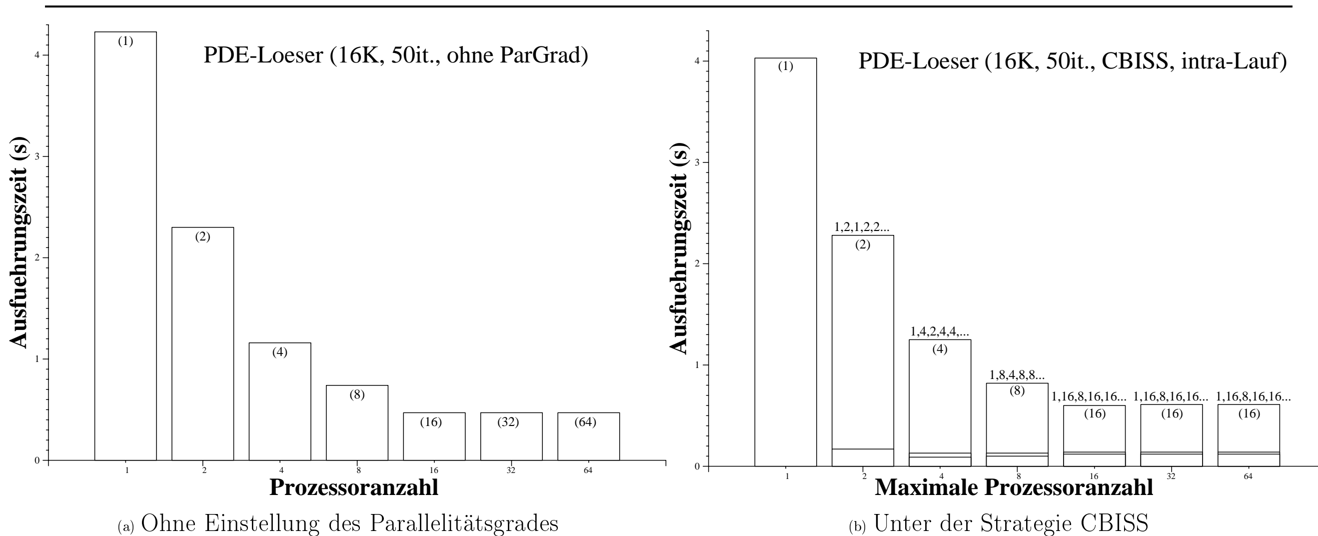


(a) Ohne Einstellung des Parallelitätsgrades (b) Unter der Strategie CBISS (c) Nach der Einstellung

Abbildung 7.36: Nutzen-Kosten-Relation des PDE-Lösers unter der Strategie CBISS

Abbildung 7.37a zeigt das Verhalten der Ausführungszeit des Programmes bei einer wachsenden Prozessoranzahl und Problemgröße gleich 16K, 50 Iterationen ohne Einstellung des Parallelitätsgrades, während Abbildung 7.37b das Ergebnis der Intra-Lauf-Einstellung darstellt. Eine Einstellung beispielsweise mit einer maximalen Prozessoranzahl gleich 64 ergibt den Parallelitätsgrad 16 (siehe Abbildung 7.36). Dadurch wurde die maximale Nutzen-Kosten-Relation erreicht, und die Ausführungszeit um ca. 36% erhöht. Die Nutzen-Kosten-

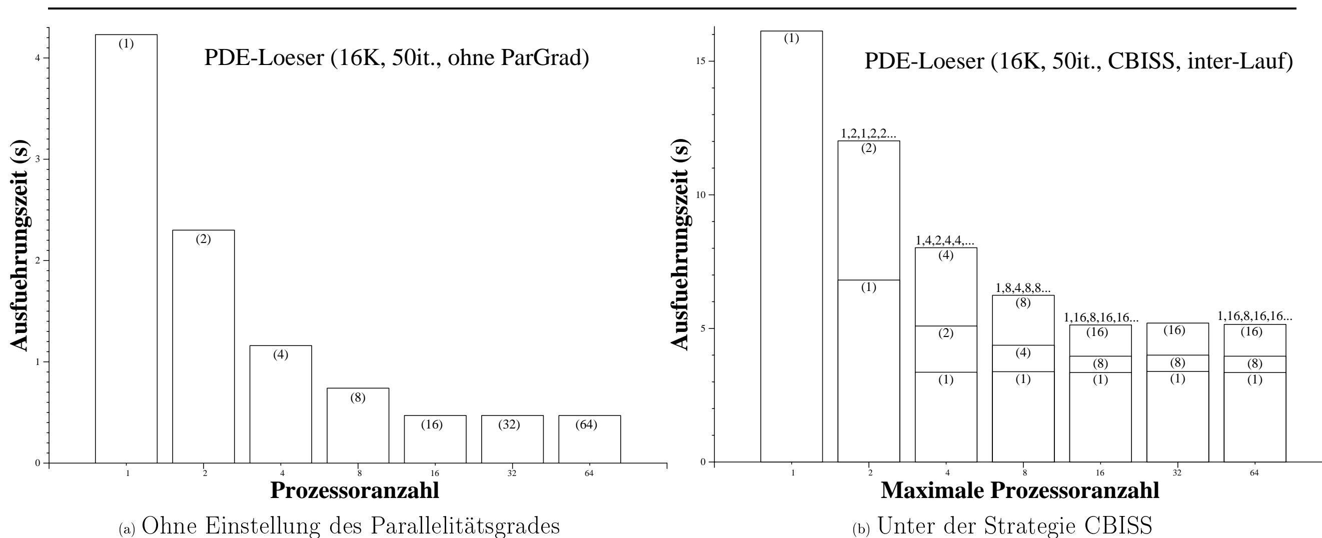
Relation wurde dabei um ca. Faktor 8 erhöht.



(a) Ohne Einstellung des Parallelitätsgrades (b) Unter der Strategie CBISS

Abbildung 7.37: PDE-Löser bei einer Intra-Lauf-Einstellung unter der Strategie CBISS

Bei einer Inter-Lauf-Einstellung wurde dieses Ergebnis bestätigt. Abbildung 7.38a stellt das Verhalten der Ausführungszeit des Programmes bei einer wachsenden Prozessoranzahl und Problemgröße gleich 16K, 50 Iterationen ohne Einstellung des Parallelitätsgrades dar, während Abbildung 7.38b das Ergebnis der Inter-Lauf-Einstellung des Parallelitätsgrades zeigt. Über sechs Iterationen wurde der Parallelitätsgrad beispielsweise von 64 auf 16 richtig eingestellt, wobei dadurch die maximale Nutzen-Kosten-Relation erreicht wurde. Die Nutzen-Kosten-Relation wurde durch die Einstellung um ca. Faktor 8 erhöht.



(a) Ohne Einstellung des Parallelitätsgrades (b) Unter der Strategie CBISS

Abbildung 7.38: PDE-Löser bei einer Inter-Lauf-Einstellung unter der Strategie CBISS

7.3.5 Genauigkeit des Suchverfahrens

Wie in Abschnitt 5.4 detailliert erläutert wurde, handelt es sich bei dem Suchverfahren zur Bestimmung des Parallelitätsgrades um eine vereinfachte Binärsuche. Hier wird die Genauigkeit der durch das Suchverfahren gelieferten Ergebnisse im Vergleich zu den realen optimalen Parallelitätsgraden diskutiert. Die realen optimalen Parallelitätsgrade wurden durch Messungen auf der Cray T3E/900 bestimmt.

Tabelle 7.16: Vergleich der Ergebnisse vom Suchverfahren mit dem realen Optimum

Anwendung	optimaler Parallelitätsgrad		Fehler
	Suchverfahren	Realer Wert	
Ziel: Minimierung der Ausführungszeit			
Radix-Sortierung	4	4	0,0%
LL1	64	64	0,0%
LL3	32	32	0,0%
LL13	64	64	0,0%
Veltran-Operator	64	64	0,0%
TLM-Verfahren	64	64	0,0%
PDE-Löser	16	16	0,0%
Ziel: Einhaltung einer Effizienzschranke			
Radix-Sortierung	2	2	0,0%
LL1	16	16	0,0%
LL3	8	8	0,0%
LL13	4	4	0,0%
Veltran-Operator	16	16	0,0%
PDE-Löser	8	8	0,0%
Ziel: Maximierung der Nutzen-Kosten-Relation			
Radix-Sortierung	1	1	0,0%
LL1	32	32	0,0%
LL3	32	32	0,0%
LL13	2	2	0,0%
Veltran-Operator	64	64	0,0%
TLM-Verfahren	32	32	0,0%
PDE-Löser	16	16	0,0%

Tabelle 7.16 zeigt den o.g. Vergleich anhand der Ergebnisse für einige der untersuchten Fälle mit vier unterschiedlichen Strategien, nämlich RISS, PeSS, EISS und CBISS (siehe Abschnitt 5.5). Die erste Spalte präsentiert die Anwendung, die zweite und dritte Spalte den über das Suchverfahren bestimmten bzw. den realen optimalen Parallelitätsgrad. Die letzte Spalte gibt den Fehler an, der durch die Auswirkung einer falschen Einstellung des Parallelitätsgrades verursacht wird (d.h. der über das Suchverfahren bestimmte und der reale optimale Parallelitätasgrad stimmen nicht überein). Dieser Fehler bezieht sich, je nach angewendeter Strategie zur Einstellung des Parallelitätsgrades, auf die Ausführungszeit, Effizienz oder Nutzen-Kosten-Relation des parallelen Programmes. In allen untersuchten Fällen stimmte der reale mit dem über das Suchverfahren ermittelten optimalen Parallelitätsgrad überein, obwohl in Tabelle 7.16 nur ein Fall mit jeweils einer Problemgröße für jedes Programm dargestellt wird. Dies deutet darauf hin, daß die Vereinfachung des Suchverfahrens

zur Bestimmung des Parallelitätsgrades die Genauigkeit der gelieferten Ergebnisse nicht beeinträchtigt hat.

7.3.6 Extrapolation aus den erzielten Ergebnissen

Es wäre natürlich ideal, wenn man die für die sieben untersuchten Programmen erzielten Ergebnisse verallgemeinern könnte und dadurch für ein neues, unbekanntes paralleles Programm eine Aussage der folgenden Art treffen könnte: *„Es ist sinnvoll, für das Programm A die Strategie X anzuwenden“*. Solch eine Aussage zu treffen ist aber nicht möglich, allein wegen der unterschiedlichen Ziele, die bei der Einstellung des Parallelitätsgrades je nach Strategie verfolgt werden. Man muß im voraus wissen, warum man den Parallelitätsgrad einstellen möchte: möchte man die Ausführungszeit minimieren, eine Effizienzschranke einhalten oder die Nutzen-Kosten-Relation maximieren?

Wenn man weiß, nach welcher Strategie man den Parallelitätsgrad eingestellt haben möchte, wäre eine Aussage dieser Art angebracht: *„Die Einstellung des Parallelitätsgrades für das Programm A nach der Strategie X wird ihr Ziel erreichen“*. Solch eine allgemeine Aussage kann aber erst dann getroffen werden, wenn man verschiedene Charakteristiken des Programmes kennt bzw. berücksichtigt. Diese Charakteristiken können in allgemeine und Strategie-spezifische Charakteristiken unterteilt werden. Allgemeine Charakteristiken sind:

- Mehraufwand der Einstellung des Parallelitätsgrades: Diese Charakteristik bezieht sich auf die für die Einstellung des Parallelitätsgrades benötigten Messungen sowie auf den Umfang der bei einer Änderung des Parallelitätsgrades benötigten Datenumverteilung.

- Problemgröße: Diese Charakteristik betrachtet sowohl die Granularität als auch die Anzahl der Iterationen der parallelisierbaren Schleife, deren Parallelitätsgrad optimiert werden soll. Die *Granularität der Schleife* umfaßt die Menge der Operationen, die es in einer parallelisierbaren Schleife pro Durchlauf zu berechnen gibt.

Die Granularität und die Anzahl der Iterationen der Schleife geben an, inwiefern der Mehraufwand der Einstellung des Parallelitätsgrades amortisiert werden kann. Wenn bei einer grobgranular parallelisierbaren Schleife, deren Gesamtanzahl der Iterationen sich beispielsweise im Bereich von Tausenden befindet, der optimale Parallelitätsgrad nach 10 Iterationen gefunden werden konnte, fällt der Mehraufwand der Einstellung nicht mehr ins Gewicht. Zusammengefaßt läßt sich sagen, daß das Verhältnis zwischen der Problemgröße und dem Mehraufwand entscheidet, ob es sich lohnt, den Parallelitätsgrad eines bestimmten Programmes einzustellen.

Die Strategie-spezifischen Charakteristiken eines Programmes beziehen sich auf bestimmte Kennzahlen und werden im folgenden erläutert:

- Die Ausführungszeit des Programmes erreicht ihren minimalen Wert bei einer Prozessoranzahl, die kleiner ist als die Gesamtanzahl der benutzten Prozessoren, und fängt wieder an, zu steigen. Folge daraus ist, daß die Ausführungszeit des Programmes minimiert werden kann. Diese Charakteristik bezieht sich auf die Strategien RISS und PeSS.

- Die Effizienz steht unterhalb eines Grenzwertes und kann verbessert werden, indem die Anzahl der benutzten Prozessoren reduziert wird. Diese Charakteristik bezieht sich auf die Strategie EISS.

- Die Nutzen-Kosten-Relation steigt bis zu einem maximalen Wert und fängt an, zu sinken, wenn die Prozessoranzahl kleiner ist als die Gesamtanzahl der benutzten Prozessoren. Daraus folgt, daß die Nutzen-Kosten-Relation maximiert werden kann. Diese Charakteristik bezieht sich auf die Strategie CBISS.

Für die Inter-Lauf-Einstellungen sind die allgemeinen Charakteristiken nicht von Bedeutung, weil erstens keine Datenumverteilung vorgenommen werden muß, und zweitens weil die Problemgröße keine Rolle spielt, da der Parallelitätsgrad in Vorläufen des Programmes bestimmt wird. Die Inter-Lauf-Einstellung des Parallelitätsgrades eines Programmes erreicht ihr Ziel immer dann, wenn die entsprechende Strategie-spezifische Charakteristik vorhanden ist. Auf der anderen Seite müssen sowohl die allgemeinen Charakteristiken als auch die entsprechende Strategie-spezifische Charakteristik stimmen, um eine Intra-Lauf-Einstellung zu ermöglichen.

Hier werden noch Möglichkeiten für verkettete Schlußfolgerungen anhand der Strategie-spezifischen Charakteristiken betrachtet. Im allgemeinen läßt sich sowohl für die Inter-Lauf- als auch für die Intra-Lauf-Einstellung feststellen, daß für parallele Programme, deren Ausführungszeit unter der Strategie RISS bzw. PeSS minimiert werden konnte, die Nutzen-Kosten-Relation unter der Strategie CBISS maximiert und die Effizienz unter der Strategie EISS auch verbessert werden können. Das Umgekehrte ist aber nicht gewährleistet, d.h. wenn die Nutzen-Kosten-Relation maximiert bzw. die Effizienz verbessert werden kann, ist es nicht offensichtlich, daß sich die Ausführungszeit minimieren läßt. Dies hängt hauptsächlich von der entsprechenden Strategie-spezifischen Charakteristik des betrachteten parallelen Programmes ab. Auch hier müssen bei einer Intra-Lauf-Einstellung die allgemeinen Charakteristiken noch zusätzlich mit einbezogen werden. Angenommen, daß sich die Ausführungszeit minimieren läßt, würde man sich beispielsweise noch die Frage stellen, ob der Mehraufwand der Einstellung des Parallelitätsgrades über die Anzahl der Iterationen der parallelisierbaren Schleife amortisiert werden kann.

7.4 Vergleich der zwei Methoden

Hier werden die zwei im Rahmen dieser Arbeit entwickelten Methoden zur Bestimmung des optimalen Parallelitätsgrades in Hinblick auf den dabei entstandenen Fehler, die Verfügbarkeit des optimalen Parallelitätsgrades, die Anzahl der allozierten Prozessoren, die Notwendigkeit der Aufwandsabschätzung sowie die Art der Einstellung verglichen. Zudem werden Vor- und Nachteile beider Methoden diskutiert.

Anhand von sechs Beispielen, nämlich dem Radix-Sortieralgorithmus, dem Programm LL1 und LL13, dem Veltran-Operator, dem TLM-Verfahren sowie dem PDE-Löser, werden die bei der Bestimmung des Parallelitätsgrades entstandenen Abweichungen gegenübergestellt (siehe Tabelle 7.17). Das Programm LL3 wurde hier wegen der ungenauen Angabe seines Kommunikationsaufwandes (siehe 7.2.3.6) nicht berücksichtigt. Die erste Spalte in Tabelle 7.17 gibt die jeweilige Beispiel-Anwendung an. Die drei darauffolgenden Spalten stellen die Fehler dar, die durch die Auswirkung der Abweichung zwischen den ggf. unterschiedlichen Parallelitätsgraden auf die Ausführungszeit, die Effizienz bzw. die Nutzen-Kosten-Relation zustande kommen. Die Angabe der Fehler handelt sich um über verschiedene Problemgrößen berechnete Durchschnittswerte.

Die erste Fehler-Spalte zeigt die Fehler zwischen den Ausführungszeiten, den Effizienzen und den Nutzen-Kosten-Relationen, die aus der Abweichung zwischen dem durch das mathematische Modell und dem durch das Suchverfahren gelieferten optimalen Parallelitätsgrad

Tabelle 7.17: Vergleich der Ergebnisse der zwei Methoden zur Bestimmung des optimalen Parallelitätsgrades

Anwendung	$Fehler_1$[a]	$Fehler_2$[b]	$Fehler_3$[c]
Ziel: Minimierung der Ausführungszeit			
Radix-Sortierung	1,3%	1,3%	0,0%
LL1	11,5%	11,5%	0,0%
LL13	0,0%	0,0%	0,0%
Veltran-Operator	0,0%	0,0%	0,0%
TLM-Verfahren	40,9%	40,9%	0,0%
PDE-Löser	0,0%	0,0%	0,0%
Ziel: Einhaltung einer Effizienzschranke			
Radix-Sortierung	96,0%	96,0%	0,0%
LL1	3,4%	3,4%	0,0%
LL13	15,7%	15,7%	0,0%
Veltran-Operator	18,9%	18,9%	0,0%
TLM-Verfahren	11,7%	11,7%	0,0%
PDE-Löser	21,0%	21,0%	0,0%
Ziel: Maximierung der Nutzen-Kosten-Relation			
Radix-Sortierung	0,0%	0,0%	0,0%
LL1	5,1%	5,1%	0,0%
LL13	3,2%	3,2%	0,0%
Veltran-Operator	0,0%	0,0%	0,0%
TLM-Verfahren	0,0%	0,0%	0,0%
PDE-Löser	0,0%	0,0%	0,0%

[a] Stellt den Fehler zwischen den Ausführungszeiten, den Effizienzen bzw. den Nutzen-Kosten-Relationen dar, der aus der Abweichung zwischen den optimalen Parallelitätsgraden resultiert, die einerseits über das mathematische Modell und andererseits über das Suchverfahren bestimmt wurden.

[b] Stellt den Fehler zwischen den Ausführungszeiten, den Effizienzen bzw. den Nutzen-Kosten-Relationen dar, der aus der Abweichung zwischen dem über das mathematische Modell berechneten und dem realen optimalen Parallelitätsgrad resultiert.

[c] Stellt den Fehler zwischen den Ausführungszeiten, den Effizienzen bzw. den Nutzen-Kosten-Relationen dar, die anhand des über das Suchverfahren gefundenen optimalen Parallelitätsgrades bzw. des realen optimalen Parallelitätsgrades berechnet wurden.

resultieren. Der zweite Fehler bezieht sich auf die Auswirkung der Abweichung zwischen dem durch das mathematische Modell ermittelten und dem realen optimalen Parallelitätsgrad. Die dritte Fehler-Spalte stellt die Fehler dar, die aus der Abweichung zwischen dem über das Suchverfahren bestimmten und dem realen optimalen Parallelitätsgrad resultiert.

Anhand der Tabelle 7.17 kann festgestellt werden, daß die durch das mathematische Modell bzw. durch das Suchverfahren erzielten Ausführungszeit, Effizienz bzw. Nutzen-Kosten-Relation in 33,3% der untersuchten Fälle nicht voneinander abweichen, in 61,1% der Fälle weichen sie nur bis zu 10% voneinander ab, und in 83,3% der untersuchten Fälle ist die Abweichung kleiner als 20%. Weiterhin ist zu beobachten, daß der Fehler zwischen dem mathematisch bestimmten und dem realen Wert im Durchschnitt ca. 12% beträgt, während der Fehler zwischen dem mittels des Suchverfahrens ermittelten und dem realen Wert Null ist. Für eine detaillierte, anwendungsspezifische Diskussion der untersuchten Fälle wird auf den Abschnitt 7.2.3 verwiesen.

Tabelle 7.18: Vergleich zwischen den zwei Methoden zur Bestimmung des Parallelitätsgrades

Kriterium	Suchverfahren	Mathematisches Modell
Verfügbarkeit des optimalen Parallelitätsgrades	dynamisch während der Laufzeit	nach der Berechnung, vor der Laufzeit
allozierte Prozessoren	Benutzer definiert	optimale Anzahl
Abschätzung des Aufwandes	keine	Berechnung, Kommunikation
Art der Einstellung	schleifenweise	Gesamtprogramm

Tabelle 7.18 stellt einen Vergleich zwischen den zwei im Rahmen dieser Arbeit entwickelten Methoden zur Bestimmung des optimalen Parallelitätsgrades zusammen. Der Einsatz des mathematischen Modells (siehe Kapitel 4) hat im Vergleich zur Bestimmung des Parallelitätsgrades über Suchverfahren (siehe Kapitel 5) drei Vorteile. Erstens wird der optimale Parallelitätsgrad über das mathematische Modell mit geringem Aufwand vor der Laufzeit des Programmes berechnet, während der optimale Parallelitätsgrad bei der Einstellung über Suchverfahren dynamisch zur Laufzeit bestimmt wird. Der zweite Vorteil ist, daß das parallele Programm von vornherein mit der optimalen Prozessoranzahl laufen wird, d.h. es werden schon am Anfang nur so viele Prozessoren alloziert wie für die Erreichung der maximalen Nutzen-Kosten-Relation benötigt werden. Für die Einstellung über Suchverfahren wird am Anfang der Ausführung des parallelen Programmes die jeweils vom Benutzer oder vom Betriebssystem festgelegte Anzahl von Prozessoren alloziert. Drittens ist die Bestimmung des optimalen Parallelitätsgrades über das mathematische Modell auf alle Parallelrechner übertragbar, für die die Latenzzeit bekannt ist.

Der Einsatz des Suchverfahrens bei der Bestimmung des optimalen Parallelitätsgrades hat zwei Vorteile. Erstens werden keine Aufwandsabschätzungen benötigt. Um das mathematische Modell anwenden zu können, werden dagegen sowohl der Berechnungs- als auch der Kommunikationsaufwand gebraucht. Der zweite Vorteil besteht darin, daß der Parallelitätsgrad beim Suchverfahren schleifenweise eingestellt wird. Beim mathematischen Modell bezieht sich die Einstellung des Parallelitätsgrades auf das gesamte Programm. Da der Parallelitätsgrad aber in der Zeit variieren kann, ist es wünschenswert, daß er u.U. in verschiedenen Phasen des Programmes neu eingestellt wird. Diese Änderung kann im mathema-

tischen Modell aber leicht vorgenommen werden, indem die Abschätzung des Berechnungs- bzw. Kommunikationsaufwandes sich nicht mehr auf das gesamte Programm sondern auf die jeweils betroffenen Schleifen bezieht. Für den Einsatz dieser Erweiterung würde aber ein Laufzeit-System (*run-time system*) benötigt werden, das die Allozierung bzw. Freigabe von Prozessoren zur Laufzeit des Programmes unterstützen würde. Das System ParGrad kann beispielsweise dafür benutzt werden, indem die über das mathematische Modell bestimmten optimalen Parallelitätsgrade in die Parallelitätsgrad-Datenbank (siehe Abschnitt 5.9) gespeichert werden.

Kapitel 8

Zusammenfassung und Ausblick

In diesem Kapitel werden die wichtigsten Beiträge dieser Arbeit nochmals zusammengefaßt. Außerdem werden auf Basis der vorliegenden Arbeit Möglichkeiten für eine weiterführende Forschung diskutiert.

8.1 Beiträge dieser Arbeit

Die zwei Thesen, die im Kapitel 1 aufgestellt wurden, konnten im Rahmen dieser Arbeit nachgewiesen werden. Diese Aussage basiert auf die verschiedenen Beiträge der vorliegenden Arbeit:

- Es ist möglich, den Parallelitätsgrad von Programmen auf Parallelrechner mit verteiltem Speicher optimal und automatisch einzustellen.

- Die Ausführungszeit, die Effizienz und die Kosten-Nutzen-Relation paralleler Programme, die für Mehrprozessorsysteme mit verteiltem Speicher entwickelt wurden, konnten durch die Einstellung des Parallelitätsgrades verbessert werden.

- Die Auslastung des Gesamtsystems wurde dadurch verbessert, daß der Parallelitätsgrad von Programmen so eingestellt wurde, daß die Effizienz sich nicht unterhalb einer vorgegebenen Schranke befand bzw. daß die Nutzen-Kosten-Relation ihren maximalen Wert erreichte.

- Als Beispiel für die zwei o.g. Punkte können einige der auf Cray T3E/900 erzielten Ergebnisse für drei Anwendungen und einen Sortieralgorithmus aufgelistet werden: Durch die Intra-Lauf-Einstellung des Parallelitätsgrades ist die Ausführungszeit des Programmes Radix-Sortierung bis zum Faktor 3,5 gesunken, die Effizienz des Veltran-Operators konnte um ca. 25% erhöht werden, die Nutzen-Kosten-Relation des TLM-Verfahrens ist um Faktor 2 gestiegen, die Inter-Lauf-Einstellung des Parallelitätsgrades über sechs Iterationen hat die Nutzen-Kosten-Relation des PDE-Lösers bis zum Faktor 8 gesteigert.

- Im Rahmen dieser Arbeit wurden zwei Methoden zur effizienten Einstellung des Parallelitätsgrades entwickelt, nämlich eine, die den Parallelitätsgrad über ein mathematisches Modell ermittelt und eine, die den Parallelitätsgrad über ein Suchverfahren bestimmt.

- Die Methode zur Einstellung des Parallelitätsgrades mittels des mathematischen Modells wurde für sieben Programme angewendet. Die daraus entstandenen Ergebnisse deuten darauf hin, daß es sich lohnt, den Parallelitätsgrad über das mathematische Modell einzustellen. Die Basis für diese Aussage sind die relativ kleinen Abweichungen (im Durchschnitt ca. 13%) in den meisten untersuchten Fällen. Die Abweichungen beziehen sich auf den Vergleich zwischen der Auswirkung der Ergebnisse dieser Methode (d.h. der durch den berechneten optimalen Parallelitätsgrad erzielten Ausführungszeit, Effizienz bzw. Nutzen-Kosten-Relation) und der Auswirkung des Einsatzes des realen, gemessenen optimalen Parallelitätsgrad.

- Mehrere Einstellungsstrategien wurden entwickelt, die unterschiedliche Kriterien berücksichtigen, um den Parallelitätsgrad über das Suchverfahren einzustellen. Solche Kriterien sind die Minimierung der Ausführungszeit eines parallelen Programmes, das Erreichen einer Effizienz, die sich nicht unterhalb einer vorgegebenen Schranke befindet, sowie die Maximierung der Nutzen-Kosten-Relation. Die entwickelten Strategien konnten zu 100% der untersuchten Fälle den Parallelitätsgrad nach den definierten Kriterien richtig einstellen.

- Ein effizienter Mechanismus wurde entwickelt, der die Datenumverteilung bei einer Änderung im Parallelitäsgrad eines parallelen Programmes vornimmt. Dieser Mechanismus ermöglicht die Einstellung des Parallelitätsgrades auf Parallelrechnern, die über verteilten Speicher verfügen. Die Datenumverteilung verschiedener Datenstrukturen wird unterstützt. Die Möglichkeit einer Erweiterung hinsichtlich benutzerdefinierter Datenstrukturen wurde berücksichtigt und ist mit wenig Aufwand vorzunehmen.

- Ein Prototyp names ParGrad wurde im Rahmen dieser Arbeit entwickelt und anhand einer Benchmark-Sammlung mit drei Anwendungen, einem Sortieralgorithmus und drei Kernen validiert. Der Prototyp ermöglicht die automatische Intra-Lauf- und Inter-Lauf-Einstellung des Parallelitätsgrades für Programme, die in C++ geschrieben sind.

8.2 Ausblick auf zukünftige Forschung

Die vorliegende Arbeit konzentriert sich auf die automatische Einstellung des Parallelitätsgrades von Programmen auf Mehrprozessorsystemen, die über verteilten Speicher verfügen. Das Ziel bzw. die angewendete Strategie einer Einstellung muß durch den Benutzer festgelegt werden. Eine **automatische Strategiewahl** würde den Benutzer von dieser Aufgabe befreien, indem über Vorläufe des betroffenen Programmes festgestellt werden würde, ob und wie das Verhalten des Programmes verbessert werden kann. Dafür würde die automatische Strategiewahl die Effizienz, die Ausführungszeit und die Kosten-Nutzen-Relation des Programmes über mehrere Vorläufe analysieren. Die Vorläufe würden über einige unterschiedliche Prozessoranzahlen vorgenommen werden, um die Entwicklung der o.g. Faktoren daraus schließen zu können. Anhand der Information über diese Entwicklung könnte eine für das Programm sinnvolle Strategie für die Einstellung des Parallelitätsgrades automatisch ausgewählt werden. Wenn das System feststellen sollte, daß es für ein paralleles Programm mehrere alternative Ziele zur Einstellung gibt, würden diese dem Benutzer zur Auswahl vorgelegt werden.

Der Benutzer des ParGrad-Systems ist für die **Festlegung der Stellen** zuständig, an denen die Einstellung des Parallelitätsgrades stattfinden soll. Diese Aufgabe könnte auch

automatisch unterstützt werden. Ein Übersetzer würde die parallelisierbaren Schleifen erkennen, bei den eine Einstellung des Parallelitätsgrades sinnvoll wäre. Dafür würde der Übersetzer die Punkte berücksichtigen, die im Abschnitt 5.3 aufgelistet wurden.

Das System ParGrad erfordert, daß alle Datenstrukturen innerhalb einer Einstellungsumgebung vom gleichen Typ sind bzw. die gleiche Art der Datenverteilung über die verschiedenen Prozessoren anwenden. Eine Erweiterung des Systems könnte die Möglichkeit vorsehen, **Datenstrukturen unterschiedlichen Typs** und mit **unterschiedlichen Arten der Datenverteilung** innerhalb derselben Einstellungsumgebung zu erlauben.

Durch eine Inter-Lauf-Einstellung kann der Parallelitätsgrad entweder von einer einzigen parallelen Schleife oder vom gesamten Programm eingestellt werden. Der Parallelitätsgrad kann aber in der Zeit variieren (siehe Abschnitt 1.1), was zum Beispiel bedeutet, daß die optimalen Parallelitätsgrade verschiedener parallelisierbaren Schleifen im Programm nicht unbedingt gleich sind. Die Folge daraus ist, daß die Inter-Lauf-Einstellung des Parallelitätsgrades an einer einzigen Stelle im Programm u.U. zu einer Verschlechterung des Verhaltens des Programmes führen kann, wenn die optimalen Parallelitätsgrade verschiedener Schleifen sehr stark voneinander abweichen. Es wäre sinnvoll, die o.g. Einschränkung zu beseitigen, indem eine **Inter-Lauf-Einstellung für mehrere parallelisierbare Schleifen** ermöglicht werden würde. Es würden, genauso wie bei einer Intra-Lauf-Einstellung, mehrere Einstellungsumgebungen zu spezifizieren sein, wobei die Anzahl der Vorläufe des Programmes für die verschiedenen Umgebungen unterschiedlich sein können.

Beim **mathematischen Modell** bezieht sich die Einstellung des Parallelitätsgrades auf das gesamte Programm. Da der Parallelitätsgrad aber in der Zeit variieren kann, ist es wünschenswert, daß er u.U. **in verschiedenen Phasen des Programmes** neu eingestellt wird. Diese Änderung kann im mathematischen Modell leicht vorgenommen werden, indem die Abschätzung des Berechnungs- bzw. Kommunikationsaufwandes sich nicht mehr auf das gesamte Programm sondern auf die jeweils betroffenen Schleifen bezieht. Für den Einsatz dieser Erweiterung würde aber ein Laufzeit-System benötigt werden, das die Allozierung bzw. Freigabe von Prozessoren zur Laufzeit des Programmes unterstützen würde. Das System ParGrad kann beispielsweise dafür eingesetzt werden, indem die über das mathematische Modell bestimmten optimalen Parallelitätsgrade in der Parallelitätsgrad-Datenbank gespeichert werden.

Eine Erweiterung des Modells, die eine Einstellung des Parallelitätsgrades mit **automatischem Übergang von einer Intra-Lauf in eine Inter-Lauf-Einstellung** vorsehen würde, könnte in Betracht gezogen werden. Wenn man feststellt, daß das Verhältnis zwischen dem Aufwand der Datenumverteilung und dem der Berechnung in der parallelen Schleife zu hoch ist, würde man zu einer Inter-Lauf-Einstellung wechseln. Dafür müßten die zwei Arten der Einstellung des Parallelitätsgrades bei der Instrumentierung des parallelen Programmes berücksichtigt werden. Dieser automatische Übergang würde dabei helfen, die Kosten der Intra-Lauf-Einstellung des Parallelitätsgrades zu amortisieren.

Um die im Rahmen dieser Arbeit **erzielten Ergebnisse verallgemeinern** zu können, sollte die hier geschilderte Methode auf die **Anwendbarkeit für verschiedene Plattformen** hin untersucht werden. Dabei wäre der Einsatz einer Standard-Bibliothek zur Kommunikation zwischen Prozessoren angebracht. Beispiele solch einer Bibliothek sind MPI (*Message Passing Interface*) und PVM (*Parallel Virtual Machine*).

Anhang A

Die *Livermore Loops* [17] sind eine Benchmark-Sammlung, die aus 24 Kernen realer Anwendungen besteht und am *Lawrence Livermore National Laboratory* entwickelt wurde. Die Kerne sind Beispiele von typischen Berechnungen in der wissenschaftlichen Datenverarbeitung. Sie werden seit den 60er Jahren für die Auswertung (*benchmarking*) von Rechnersystemen angewendet. Abgesehen von LL17 und LL20, lassen sich alle Kerne parallelisieren. Bei diesen zwei Programmen existieren Datenabhängigkeiten zwischen Schleifeniterationen, die sich nicht aufbrechen lassen. Der Kommunikationsaufwand von LL16 läßt sich nicht statisch festlegen, weil das Kommunikationsmuster von den Eingabendaten abhängt. Dieser Kern wird aus diesem Grund nicht berücksichtigt.

Tabelle A.1 basiert auf [54] und zeigt die Art des Kommunikationsaufwandes aller Programme der *Livermore Loops*. Die erste Spalte beschreibt kurz das Programm, sein Kommunikationsaufwand wird in der zweiten Spalte angegeben. Die dritte Spalte gibt die Klasse wieder, zu der das Programm gemäß der Klassifizierung in Abschnitt 4.2 gehört.

Tabelle A.1: Klassifizierung der Livermore Loops anhand des Kommunikationsaufwandes

Programm		Kommunikationsaufwand	Klasse
LL1	hydro fragment	linear	K2
LL2	ICCG excerpt	logarithmisch	K4
LL3	inner product	logarithmisch	K4
LL4	banded linear equation	logarithmisch	K2
LL5	tri-diagonal elimination	linear, logarithmisch	K2, K4
LL6	general linear recurrence equations	linear, logarithmisch	K2, K4
LL7	equation of state fragment	konstant	K1
LL8	ADI integration	konstant	K1
LL9	integrate predictors	Null	K0
LL10	difference predictors	Null	K0
LL11	first sum	logarithmisch	K4
LL12	first difference	konstant	K1
LL13	2D particle in cell	linear	K2
LL14	1D particle in cell	linear, logarithmisch	K2, K4
LL15	casual Fortran	konstant	K1
LL18	2D explicit hydrodynamics fragment	logarithmisch	K4
LL19	general linear recurrence equations	logarithmisch	K4
LL21	matrix $\times$ matrix product	logarithmisch	K4
LL22	Planckian distribution	Null	K0
LL23	2D implicit hydrodynamics fragment	logarithmisch	K4
LL24	first minimum	logarithmisch	K4

Literaturverzeichnis

[1] G. Amdahl. Validity of the single-processor approach to achieving large-scale computing capabilities. In *Proceedings of the AFIPS Conference*, page 483. AFIPS Press, 1967.

[2] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA, March 1994.

[3] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-020, NASA, December 1995.

[4] M. Berry, D. Chen, P. Koss, D. Kuck, and S. Lo. The Perfect Club benchmarks: effective performance evaluation of supercomputers. *International Journal of Supercomputer Applications*, pages 5–40, 1989.

[5] S. Brawer. *Introduction to Parallel Programming.* Academic Press, Inc., 1989.

[6] Ralph Buttler and Ewing Lusk. User's Guide to the P4 Parallel Programming System. Technical Report ANL-92/17, Argonne National Laboratory, 1992.

[7] Bruce W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leongand M. B. Monagan, and S. M. Watt. *First Leaves: A Tutorial Introduction to Maple V.* Springer-Verlag, New York and Waterloo Maple Publishing, 1992.

[8] Lyndon Clarke, Ian Glendinnig, and Rolf Hempel. The MPI Message Passing Interface Standard. Technical report, Message Passing Interface Forum, University of Tennesee, Knoxville, Tennessee, 1994.

[9] Intel Corporation. *Paragon XS/P Product Overview.* Intel Corporation, 1991.

[10] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauer, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 1–12, New York, NY, May 1993. ACM Press.

[11] David E. Culler, Andrea Dusseau, Seth C. Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, Los Alamitos, CA, November 1993. IEEE Computer Society Press.

[12] Allen B. Downey. Using Queue Time Predictions for Processor Allocation. In *Proceedings of the International Parallel Processing Symposium*, pages 35–57, Berlin, April 1997. Springer-Verlag.

[13] Derek L. Eager, John Zahorjan, and Edward Lazowska. Speedup versus Efficiency in Parallel Systems. *IEEE Transactions on Computer*, 38(3):408–423, March 1989.

[14] Guy Edjlali, Gagan Agrawal, Alan Sussman, Jim Humphries, and Joel Saltz. Compiler and Runtime Support for Programming in Adaptive Environments. Technical Report UMIACS-TR-95-83 and CS-TR-3510, UMIACS and Department of Computer Science, University of Maryland, 1997.

[15] A. Al Geist et al. PVM3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, OAK Ridge National Laboratory, 1993.

[16] A. Hori et al. Implementation of Gang Scheduling on Workstation Cluster. In *Proceedings of the 10th International Parallel Processing Symposium*, Los Alamitos, CA, April 1996. IEEE Computer Society Press.

[17] John T. Feo. An Analysis of the Computational and Parallel Complexity of the Livermore Loops. *Parallel Computing*, 7:163–185, February 1988.

[18] Peter Fischer, Martin Gebhardt, P. Argus, and Adolf J. Schwab. Parallelization of a 3D-TLM Algorithm for Different Computing Environments. In *Proceedings of the 8th International IGTE Symposium on Numerical Field Calculation in Electrical Engineering*, September 1998.

[19] Michael J. Flynn. Some Computer Organizations and their Effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, September 1972.

[20] Steven Fortune and James Wyllie. Parallelism in Random Access Machines. In *Proceedings of the 10th Symposium on Theory of Computing*, pages 114–118, New York, NY, May 1978. ACM Press.

[21] Ian T. Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1995.

[22] Martin Gebhardt. Parallelisierung des 3D-TLM-Algorithmus mittels MPI. Master's thesis, Department of Electrical Engineering, Karlsruhe University, 1998.

[23] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 120–132, New York, NY, May 1991. ACM Press.

[24] Mary Hall and Margaret Martonosi. Adaptive Parallelism in Compiler-Parallelized Code. In *Proceedings of the 2nd SUIF Compiler Workshop*, Stanford University, August 1997. USC Information Sciences Institute.

[25] Stefan U. Hänsgen. *Effiziente parallele Ausführung irregulärer rekursiver Programme.* PhD thesis, Fakultät für Informatik, Universität Karlsruhe, 1998.

[26] P. J. Hatcher, M. J. Quinn, R. J. Anderson, A. J. Lapadula, B. K. Seevers, and A. F. Bennett. Architecture-independent Scientific Programming in Dataparallel C: Three Case Studies. In *Proceedings of the 4th Annual Conference on Supercomputing*, pages 208–217, Albuquerque, NM, USA, November 1991. IEEE Computer Society Press.

[27] Hans-Ulrich Heiss. *Überlast in Rechensystemen - Modellierung und Verhinderung*. Springer-Verlag, 1988.

[28] Hans-Ulrich Heiss. *Prozessorzuteilung in Parallelrechnern*. BI-Wissenschaftsverlag, 1994.

[29] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. Series in Computer Science. McGraw-Hill Inc., 1993.

[30] Matthias Jacob. Implementing Large-Scale Parallel Geophysical Algorithms Using the Java Programming Language - A Feasibility Study. Master's thesis, Department of Informatics, Karlsruhe University, 1998.

[31] Matthias Jacob, Michael Philippsen, and Martin Karrenbach. Fallstudie: Parallele Realisierung geophysikalischer Basisalgorithmen in Java. *Informatik - Forschung & Entwicklung*, 13(2):72–78, June 1998.

[32] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. The MIT Press Cambridge, Massachusetts, London, England, 1994.

[33] Manoj Kumar. Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications. *IEEE Transactions on Computers*, 37(9):1088–1098, September 1988.

[34] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The Network Architecture of the Connection Machine CM-5. In *Symposium on Parallel and Distributed Algorithms*, June 1992.

[35] S. T. Leutenegger and M. K. Vernon. The Performance of Multiprogrammed Multiprocessor Scheduling Policies. In *Proceedings of the 1990 ACM SIGMETRICS Conference*, New York, NY, May 1990. ACM Press.

[36] Welf Löwe and Wolf Zimmermann. Upper Time Bounds for Executing PRAM-Programs on the LogP-Machine. In *Proceedings of the ACM International Conference on Supercomputing*, pages 41–50, New York, NY, July 1995. ACM Press.

[37] Honghui Lu, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. Message passing versus distributed shared memory on network workstations. In *Proceedings of the Supercomputing'95*, pages 64–65, New York, NY, December 1995. ACM Press.

[38] Udi Manber. *Introduction to Algorithms - A Creative Approach*. Addison-Wesley, 1989.

[39] Cathy McCann, Raj Vaswani, and John Zahorjan. A Dynamic Processor Allocation Policy for Multiprogrammed Shared-memory Multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, May 1993.

[40] Matthias M. Müller, Thomas M. Warschko, and Walter F. Tichy. Prefetching on the Cray-T3E: A Model and its Evaluation. Technical Report 26/97, Department of Informatics, Karlsruhe University, 1997.

[41] Wilfried Oed. Massiv-paralleles Prozessorsystem Cray T3E. Technische Dokumentation, Cray Research GmbH, München, November 1996.

[42] John K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the Third International Conference on Distributed Computing Systems*, pages 22–30, Washington, DC, USA, 1982. IEEE Computer Society Press.

[43] Kee-Hyun Park and Lawrence W. Dowdy. Dynamic Partitioning of Multiprocessor Systems. *International Journal of Parallel Programming*, 18(2):91–120, November 1989.

[44] Niels Reimer. Dynamische Einstellung des Parallelitätsgrades mit reflexiven Programmen. Master's thesis, Fakultät für Informatik, Universität Karlsruhe, 1996.

[45] 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)*, pages 313–318, Santa Barbara, CA, USA, August 1996. Springer-Verlag, LNCS 111.

[46] Steven L. Scott. Synchronization and Communication in the T3E Multiprocessor. *ACM SIGPLAN Notices*, 31(2):26–36, September 1996.

[47] Behrooz A. Shirazi. *Scheduling and load balancing in parallel and distributed systems.* IEEE Computer Society Press, 1995.

[48] Andrew Sohn and Yuetsu Kodama. Load Balanced Parallel Radix Sort. In *Proceedings of the 12th ACM International Conference on Supercomputing*, pages 312–319, New York, NY, July 1998. ACM Press.

[49] Mark S. Squillante. On the Benefits and Limitations of Dynamic Partitioning in Parallel Computer Systems. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 219–238, Berlin, April 1995. Springer-Verlag.

[50] Mark S. Squillante, Fang Wang, and Marios Papaefthymiou. An Analysis of Gang Scheduling for Multiprogrammed Parallel Computing Environments. In *Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architectures*, pages 89–98, New York, NY, June 1996. ACM Press.

[51] Andrew Tucker and Anoop Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-memory Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 159–166, New York, NY, December 1989. ACM Press.

[52] Theo Ungerer. *Parallelrechner und parallele Programmierung.* Spektrum Akadem. Verlag, 1997.

[53] Fang Wang, Marios Papaefthymiou, and Mark S. Squillante. Performance Evaluation of Gang Scheduling for Parallel and Distributed Multiprogramming. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 277–298, Berlin, April 1997. Springer-Verlag.

[54] Thomas M. Warschko. *Effiziente Kommunikation in Parallelrechnerarchitekturen*. PhD thesis, Fakultät für Informatik, Universität Karlsruhe, 1997.

[55] Steven C. Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder P. Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, New York, NY, June 1995. IEEE Computer Society Press.

[56] Kelvin K. Yue and David J. Lilja. Efficient Execution of Parallel Applications in Multiprogrammed Multiprocessor Systems. Technical Report HPPC-95-05, High-Performance Parallel Computing Research Group, Department of Electrical Engineering, Department of Computer Science, Minneapolis, Minnesota, 1995.

[57] Marco Zagha and Guy E. Blelloch. Radix Sort for Vector Multiprocessors. In *Proceedings of the Supercomputing'91*, pages 712–721, Los Alamitos, CA, November 1991. IEEE Computer Society Press.

[58] Wolf Zimmermann. *Planbare Algorithmen: Eine Methode zum maschinenunabhängigen parallelen Programmieren*. Fakultät für Informatik, Universität Karlsruhe, 1998.