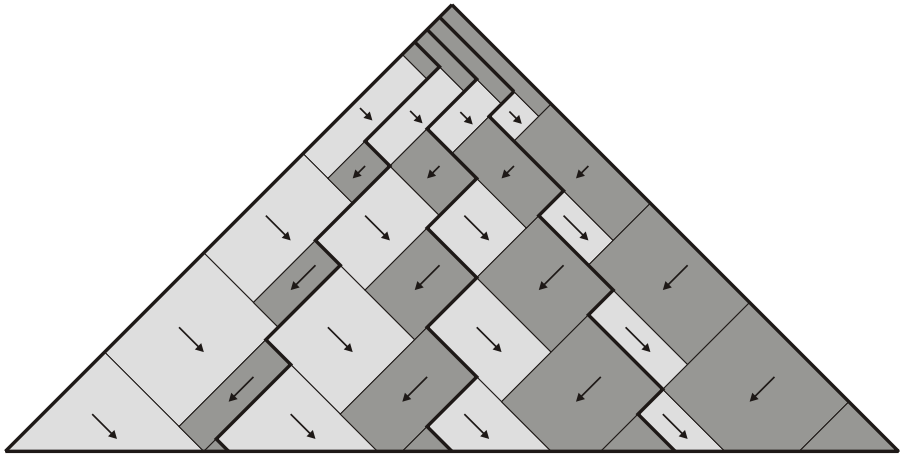
Ein universelles Lastverteilungssystem und seine Anwendung bei der Isolierung reeller Nullstellen



Dissertation

von

Thomas Decker

Schriftliche Arbeit zur Erlangung des Grades
eines Doktors der Naturwissenschaften

Fachbereich Mathematik/Informatik
Universität-Gesamthochschule Paderborn

Paderborn, im Dezember 2000

Meinen Eltern

Danksagung

Die vorliegende Arbeit wurde im Fachbereich Mathematik/Informatik an der Universität Gesamthochschule Paderborn angefertigt.

Mein besonderer Dank gilt Prof. Dr. Burkhard Monien für die Betreuung der Arbeit und für die Anleitung zum wissenschaftlichen Arbeiten. Bei meinem Kollegen Dr. Werner Krandick bedanke ich mich für die hervorragende Zusammenarbeit und die vielen hilfreichen Diskussionen. Die Arbeit an der Parallelisierung der Descartes Methode für die Isolierung reeller Nullstellen hat zahlreiche Ergebnisse in dieser Arbeit motiviert. Weiterhin bedanken möchte ich mich bei meinen Kollegen Dr. Robert Preis und Norbert Sensen für die vielen guten Ideen und die kritische Durchsicht von Teilen des Manuskriptes. Desweiteren bedanke ich mich bei Christian Voss für sein aufmerksames Korrekturlesen. Bedanken möchte ich mich auch bei Dr. Ralf Diekmann und Dr. Reinhard Lüling, die mich seit meiner Diplomarbeit durch Hilfestellungen und konstruktive Kritik unterstützt haben.

Ich möchte mich auch bei Ilja Weis bedanken, der mit seiner tatkräftigen Hilfe entscheidend dazu beigetragen hat, dass das Lastverteilungssystem VDS in seiner heutigen Form nutzbar ist. An dieser Stelle möchte ich mich bei den Mitarbeitern des PC^2, insbesondere bei Axel Keller und Andreas Krawinkel, für die Unterstützung bei der Nutzung der Parallelrechner bedanken.

Besonderen Dank verdient schließlich Bettina Lohmann für ihre geduldige und ermutigende Unterstützung während der Arbeit.

Nicht zuletzt möchte ich mich bei meinen Eltern bedanken, die mir das Studium ermöglicht und mich in jeder Hinsicht unterstützt haben.

Paderborn, im Dezember 2000 Thomas Decker

Diese Arbeit wurde gefördert durch

- den DFG-Sonderforschungsbereich 376 „Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen“,
- EC ESPRIT Long Term Research Project 20244 „ALCOM-IT“ und
- EC TMR-Grant ERB-FMGECT950052 (ICARUS 2).

Inhaltsverzeichnis

KAPITEL 1

Einleitung

"*Reality is the murder of a beautiful theory by a gang of ugly facts.*"
(Robert L. Glass, Communications of the ACM, 1996)

Heutige Betriebssysteme bieten mit ihren Diensten eine komfortable Entwicklungsumgebung für sequentielle Programme. So erlaubt beispielsweise das Konzept des virtuellen Speichers eine automatische Zuordnung von Variablen zu Speicherzellen. Durch solche automatische Zuordnungsmechanismen logischer Betriebsmittel (z.B. Variablen) zu physikalischen Betriebsmitteln (z.B. Speicherzellen) wird eine weitgehende Unabhängigkeit des Quellprogramms von der Maschine erreicht.

Im Bereich der parallelen Programmierung ist diese Unabhängigkeit im Allgemeinen nicht gegeben. Der Grund ist in den ungleich komplizierteren Zuordnungsproblemen zu suchen. Parallele Programme zeichnen sich dadurch aus, dass sie ein gegebenes Problem in Teilprobleme zerlegen, die dann von den Prozessoren des Systems bearbeitet werden. Es ergeben sich zwei Fragestellungen: Erstens, wie soll das Problem zerlegt werden? Zweitens, wie sollen die Teilprobleme den Prozessoren zugeteilt werden (Frage nach der *Lastverteilung*)?

Zunächst ist die konzeptionelle Frage nach der Art der Parallelisierung zu beantworten. Dabei kann entweder ein daten- oder ein aufgabenparallele Ansatz verfolgt werden.

Bei einem datenparallelen Ansatz wird das zu lösende Problem durch eine Menge von Datenobjekten beschrieben, auf denen jeweils dieselben Operationen ausgeführt werden. Die Lösung von partiellen Differenzialgleichungen durch numerische Simulationen führt beispielsweise zu einer Beschreibung des Problems durch einen Graphen. Die Knoten des Graphen repräsentieren die Datenobjekte und die Kanten die Abhängigkeiten zwischen den Datenobjekten. Da dieser Graph ein hohes Maß an Lokalität aufweist, kann er sehr gut in kompakte Gebiete aufgeteilt werden, die weitgehend unabhängig voneinander bearbeitet werden können.

Beim aufgabenparallelen Ansatz wird das Ausgangsproblem in Teilaufgaben zerlegt, die durch eine Reihe von Parametern beschrieben werden. Jedem dieser Parameter wird entweder initial ein Wert zugeordnet, oder er bekommt seinen Wert während der parallelen Berechnung nach der Bearbeitung anderer Teilaufgaben. Im ersten Fall wird ein Parameter als *gebunden* und im zweiten als *offen* bezeichnet. Sobald alle offenen Parameter einer Teilaufgabe einen definierten Wert haben, kann sie bearbeitet werden. Während der Bearbeitung können neue Teilaufgaben definiert werden.

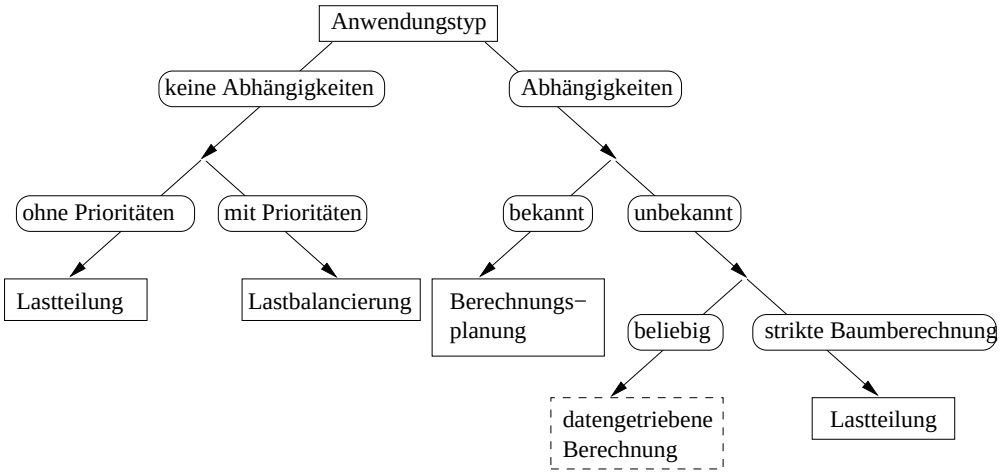


ABBILDUNG 1.1.1. Unterschiedliche Anwendungstypen verlangen den Einsatz unterschiedlicher Lastverteilungstechniken.

Eine eingehende Betrachtung des Lastverteilungsproblems für datenparallele Anwendungen findet sich in der Dissertation Diekmanns [**42**]. Bei den dort betrachteten Anwendungen hängt die Berechnung eines Datenobjektes von einer (kleinen) Menge benachbarter Datenobjekte ab. Ein Datenobjekt ist somit auf dem Prozessor am effizientesten zu verarbeiten, auf dem sich seine benachbarten Datenobjekte befinden. Diese besondere Eignung eines bestimmten Prozessors für die Bearbeitung eines Datums wird als *Affinität* bezeichnet. Die bei der Verteilung von datenparallelen Anwendungen zum Einsatz kommenden Lastverteilungstechniken zeichnen sich dadurch aus, dass sie die Affinitäten berücksichtigen.

Auch bei aufgabenparallelen Anwendungen können Affinitäten vorliegen. Ein typisches Beispiel sind verteilte Datenbanken, da eine Transaktion jeweils von dem Server am effizientesten bearbeitet werden kann, der den schnellsten Zufgriff auf die betroffenen Daten hat. In dieser Arbeit werden Anwendungen betrachtet, bei denen Affinitäten nicht oder nur in geringem Maße auftreten. Beispiele solcher Anwendungen sind paralleles Branch&Bound und reine Divide&Conquer Algorithmen.

Die Klasse der aufgabenparallelen Anwendungen kann nach unterschiedlichen Kriterien in *Anwendungstypen* unterteilt werden. Es zeigt sich, dass die Beantwortung der Frage nach der Lastverteilung vom Typ der Anwendung abhängt. Im folgenden Abschnitt werden diese Zusammenhänge im Einzelnen diskutiert.

1.1. Parallele Programmierparadigmen

Aufgabenparallele Anwendungen können durch einen gerichteten azyklischen Graphen (*Aufgabengraphen, DAG*) $G = (V, S \cup D)$ beschrieben werden. Hierbei ist V die Menge der Teilaufgaben. Die Kantenmenge S repräsentiert die Abhängigkeiten, die durch die Generierung der Teilaufgaben entstehen. Eine Kante (u, v) ist genau dann in der Menge S enthalten, wenn die Aufgabe v von u generiert wird. Die Kantenmenge D bezeichnet die Datenabhängigkeiten zwischen den Aufgaben. Es existiert eine Kante (u, v) genau dann in D, wenn u einen Wert berechnet, der einem offenen Parameter von v zugewiesen wird.

Die Klasse der aufgabenparallelen Anwendungen lässt sich durch Einschränkungen der zulässigen Graphen in Unterklassen aufteilen. Abbildung 1.1.1 zeigt eine solche Klassifizierung.

Betrachten wir zunächst Anwendungen, die sich in voneinander unabhängige Teilaufgaben zerlegen lassen ($D = \emptyset$, linker Ast in Abbildung 1.1.1). Beispiele für solche Anwendungen sind paralleles Branch&Bound und Anwendungen, die das *Farming*-Paradigma nutzen. Bei der Verwendung des Farming-Paradigmas geht man im einfachsten Fall davon aus, dass einen Prozess (der *Farmer*) Aufgaben generiert, die von den Arbeiterprozessen bearbeitet werden. Nach der Bearbeitung der Aufgaben werden die Ergebnisse an den Farmer zurückgeliefert. Falls es mehrere Farmer gibt, sprechen wir von *verteiltem Farming.* Um bei derartigen Anwendungen ein paralleles System optimal zu nutzen, muss bei der Lastverteilung dafür gesorgt werden, dass alle Prozessoren möglichst ununterbrochen an Aufgabenknoten arbeiten. Diese Art der Lastverteilung wird als *Lastteilung* (*load-sharing*) bezeichnet. Im Gegensatz dazu verfolgen *Lastbalancierungsmethoden* das Ziel, die Last aller Prozessoren zu jedem Zeitpunkt auf dem gleichen Wert zu halten.

Eine Balancierung der Last ist unter anderem dann erforderlich, wenn jede Teilaufgabe mit einer Priorität versehen ist. Ein gutes Beispiel dafür ist paralleles best-first Branch&Bound. Eine weit verbreitete Technik, Branch&Bound Anwendungen zu parallelisieren, besteht darin, jeden Prozessor einen sequentiellen Branch&Bound Algorithmus ausführen zu lassen [**96**]. Die Prozessoren arbeiten ausschließlich an den ihnen zugewiesenen Teilproblemen. Dabei starten zunächst alle Prozessoren mit einem unterschiedlichen Teilproblem (*initiale Partitionierung*). Sobald eine neue Schranke gefunden wird, wird sie an alle Prozessoren geschickt. Das Empfangen einer neuen Schranke hat zur Folge, dass alle Teilprobleme mit einer schlechteren Schranke gelöscht werden.

Die initiale Partitionierung kann aus zwei Gründen zu einer schlechten Systemauslastung führen. Erstens kann die Arbeit, die in unterschiedlichen Teilen des Suchraums entsteht, nur schwer von vornherein abgeschätzt werden. Diese *quantitative Unbalanciertheit* kann zu unbeschäftigten Prozessoren führen. Zweitens können sich die unterschiedlichen Teile in ihrer Qualität oder in der Anzahl guter Lösungen unterscheiden (*qualitative Unbalanciertheit*). Da der sequentielle Branch&Bound Algorithmus die Teilpobleme in einer strikten best-first Reihenfolge bearbeitet, können qualitative Unbalanciertheiten dazu führen, dass der parallele Branch&Bound Algorithmus Teilprobleme bearbeitet, die vom sequentiellen Algorithmus nicht bearbeitet werden.

Um der quantitativen Unbalanciertheit entgegenzuwirken, reicht der Einsatz eines Lastteilungsverfahrens aus. Für die Vermeidung qualitativer Unbalanciertheiten ist es jedoch notwendig, die Prozessoren entsprechend der Qualität der Teillösungen zu balancieren. Auf diese Weise wird gewährleistet, dass alle Prozessoren an „gleich wichtigen“ Teilproblemen arbeiten. Eine weitere Möglichkeit, beide Arten der Unbalanciertheit zu vermeiden, besteht darin, die Last eines Prozessors in Abhängigkeit von Anzahl und Qualität der auf ihm platzierten Teilprobleme zu definieren, und die Last der Prozessoren entsprechend dieses Maßes zu balancieren. Diese Problematik wurde unter anderem in der Dissertation Lülings ausführlich behandelt [**96**].

Falls die Teilaufgaben Datenabhängigkeiten aufweisen ($D \neq \emptyset$, rechter Ast in Abbildung 1.1.1), ist zunächst zu unterscheiden, ob diese Abhängigkeiten bereits vor der parallelen Berechnung bekannt sind. Sind sie bekannt, verlieren die Abhängigkeiten, die aus der Aufgabengenerierung entstehen, an Bedeutung, und wir können $S = \emptyset$ annehmen. In diesem Fall spechen wir

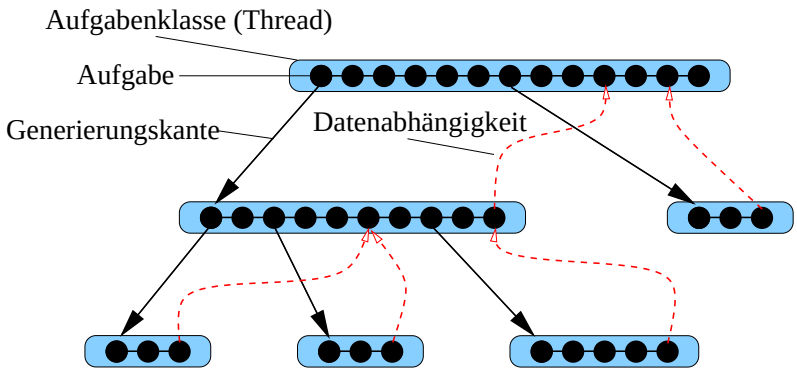


ABBILDUNG 1.1.2. Eine strikte Baumberechnung. Die schattierten Rahmen zeigen die einzelnen Aufgabenklassen (*Threads*). Die durchgezogenen Pfeile zwischen den Threads veranschaulichen die Generierungskanten und die gestrichelten Pfeile die Datenabhängigkeiten.

von *statischen Datenflussberechnungen.* Durch Verwendung von *Planungsalgorithmen* für Aufgabengraphen (*task graph scheduling*) kann eine Prozessorzuteilung vor der Ausführung des Programmes berechnet werden. Insbesondere wenn der Aufgabengraph eingabeunabhängig ist, und viele Läufe durchgeführt werden sollen, kann es sich lohnen, aufwendige Planungsalgorithmen einzusetzen, die im Allgemeinen zu deutlich besseren Ergebnissen führen als Lastverteilungsalgorithmen, die die Zuordnung während der Laufzeit bestimmen.

Falls die Datenabhängigkeiten nicht bekannt sind, müssen sie während der Laufzeit aufgelöst werden. Ferner muss die Prozessorzuordnung während der Laufzeit vorgenommen werden. Hierbei ist es wie bei den datenparallelen Anwendungen notwendig, bei der Lastteilung die Affinitäten, die durch die Datenabhängigkeiten entstehen, zu berücksichtigen.

Beide Vorgänge, die Auflösung der Datenabhängigkeiten und die Lastverteilung, können erheblich vereinfacht werden, wenn gewisse Struktureigenschaften der Datenflussgraphen bekannt sind. Da Bäume wegen ihrer vielfältigen Anwendungen von besonderem Interesse sind, wird die folgende Klasse betrachtet:

DEFINITION 1.1.1. Ein Aufgabengraph $G = (V, S \cup D)$ heißt *strikte Baumberechnung*, wenn eine Äquivalenzrelation $\equiv$ auf V existiert, so dass die folgenden Bedingungen erfüllt sind:

(1) Die durch die Äquivalenzklassen definierten Teilgraphen sind jeweils isomorph zu einem Pfad.
(2) Der Graph $G' = (V, S)$ ist isomorph zu einem Baum.
(3) Für alle Datenabhängigkeiten $(u, v) \in D$ gilt, dass $u \not\equiv v$, und $\exists_{v', u' \in V,\, v \equiv v',\, u \equiv u'} (v', u') \in S$.

In Abbildung 1.1.2 ist ein Beispiel einer strikten Baumberechnung dargestellt. Die Äquivalenzklassen bilden parallele Kontrollflüsse, die oft als *Threads* bezeichnet werden. Wir sagen, dass eine Kante aus der Kantenmenge E zwischen zwei Threads A und B existiert, wenn es eine Kante $(u, v) \in E$ mit $u \in A$ und $v \in B$ gibt. Die Threads bilden bezüglich S einen *Aufrufbaum.* Eine wichtige Unterklasse dieser Berechnungen sind die *Fork-Join*-Berechnungen. Sie unterscheiden sich insofern von strikten Baumberechnungen, dass es zwischen zwei Threads höchstens eine

Kante aus D gibt. Die Klasse der Fork-Join-Berechnungen umfasst beispielsweise die der Divide&Conquer Algorithmen. Generell konnte für die Lastverteilung von strikten Baumberechnungen gezeigt werden, dass die Verwendung einer *Work-Stealing*-Technik, bei der unbeschäftigte Prozessoren von beschäftigten Prozessoren jeweils einen möglichst weit oben im Aufrufbaum stehenden Thread „stehlen“, zu aymptotisch optimalen Ergebnissen führt [**9**].

Wir haben gesehen, dass Einschränkungen der Aufgabengraphen zu verschiedenen Anwendungstypen führen, die jeweils andere Parallelisierungstechniken erfordern. Dadurch ergeben sich Anforderungen an ein allgemein verwendbares Lastverteilungssystem, auf die im Folgenden näher eingegangen werden wird.

1.2. Anforderungen an ein universelles Lastverteilungssystem

In den letzten Jahren wurde eine Vielzahl von Systemen vorgestellt, die die Lastverteilung in parallelen Anwendungen übernehmen. Analog zur Deklaration von Variablen muss in den parallelen Programmen spezifiziert werden, woraus die einzelnen Teilaufgaben bestehen, und in welcher Beziehung sie zueinander stehen. Die Systeme unterscheiden sich vor allem darin, auf welche Weise die Aufgaben beschrieben werden, und welches Programmierparadigma sie unterstützen. Beispiele sind die Cilk Programmiersprache für strikte Baumberechnungen, die DOTS-Bibliothek für Fork-Join-Berechnungen oder das RAPID System für statische Datenflussberechnungen.

Die meisten Systeme legen den Programmierer auf ein bestimmtes Paradigma fest. Auf Grund fehlender Kompatibilität dieser Systeme ist es nicht möglich, verschiedene Paradigmen in derselben Anwendung zu nutzen. Diese Tatsache bereitet vor allem dann Schwierigkeiten, wenn Anwendungen aus mehreren Parallelitätsebenen bestehen. Als Beispiel sei ein Divide&Conquer Algorithmus genannt, dessen Teilaufgaben parallel bearbeitet werden können. Ein solcher Algorithmus verfügt über zwei Parallelitätsebenen: der Baumparallelität, die durch die gleichzeitige Berechnung mehrerer Teilaufgaben entsteht, und der Knotenparallelität, die durch die parallele Berechnung jeder einzelnen Aufgabe entsteht. Vor allem wenn der Aufgabenbaum nur schmal ist, kann durch die Ausnutzung der Knotenparallelität die Laufzeit verkürzt werden. Wünschenswert ist somit die Ausnutzung der Parallelität auf beiden Ebenen. Die Festlegung auf ein Paradigma bedingt, dass dieses nicht möglich ist, wenn die Parallelisierung der beiden Ebenen auf unterschiedlichen Paradigmen beruht. Selbst wenn beide Ebenen mit demselben Paradigma parallelisiert werden können, kann die mit den Systemen einhergehende Einschränkung auf eine einheitliche Lastverteilungstechnik zu ineffizienten Parallelisierungen führen.

Ein *universell* einsetzbares Lastverteilungssystem sollte daher

- mehrere Paradigmen unterstützen,
- für jedes dieser Paradigmen effiziente Lastverteilungstechniken zur Verfügung stellen, und
- es ermöglichen, verschiedene Paradigmen und Lastverteilungstechniken zu kombinieren.

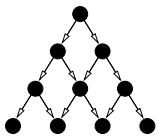
1.3. Resultate und Gliederung der Arbeit

Im Mittelpunkt der Arbeit steht das universelle Lastverteilungssystem *„Virtual Data Space“* (*VDS*). Das System integriert sowohl bekannte Lastverteilungsverfahren als auch neue, die in dieser Arbeit vorgestellt werden. Bevor in Kapitel 6 auf die Konzeption des Systems eingegangen wird, werden in Kapitel 2 Grundlagen für die Beurteilung der Skalierbarkeit eines parallelen

Algorithmus eingeführt. In den folgenden Kapiteln 3 und 4 werden zwei statische Planungsstrategien vorgestellt, und in Kapitel 5 wird die statische Lastbalancierung unabhängiger Aufgaben diskutiert. In Kapitel 7 wird die Parallelisierung der Descartes Methode für die Isolierung reeller Nullstellen mit Hilfe von VDS vorgestellt.

In den Analysen wird die Dominanznotation nach Collins [**26**] benutzt: Wenn zwei reellwertige Funktionen f und g auf einem gemeinsamen Wertebereich S definiert sind, wird f von g *dominiert* ($f \preceq g$), wenn es eine positive Zahl c gibt mit $f(x) \leq c\, g(x)$ für alle $x \in S$. Wenn $f \preceq g$ und $g \preceq f$, sagen wir f ist *kodominant* mit g und schreiben $f \sim g$.

In Kapitel 2 wird das von Kumar eingeführte Konzept der *Isoeffizienz* formal definiert, und es werden einige grundlegende Eigenschaften, die sich aus dieser Definition ergeben, hergeleitet. Mit Hilfe dieser Definition kann die Isoeffizienz von Algorithmen angegeben werden, deren Effizienz nicht ausschliesslich von der sequentiellen Rechenzeit abhängt. Es zeigt sich, dass die Klasse der Probleme, für die es einen Algorithmus mit polynomieller Isoeffizienz gibt, eine echte Unterklasse der von Kruskal, Rudolph und Snir eingeführten Klasse EP ist [**88**].



In Kapitel 3 werden statische Planungsverfahren für den sogenannten *Pyramidengraphen* diskutiert. Eine Pyramide der Höhe n besteht aus den Knoten $(i,\ j)$ mit $0 \leq i,\ j < n$ und $i + j < n$. Jeder dieser Knoten benötigt für seine Berechnung die Ergebnisse der Vorgängerknoten $(i-1,\ j)$ und $(i,\ j-1)$. Derartige Abhängigkeiten werden unter anderem von Algorithmen erzeugt, die auf dynamischer Programmierung beruhen (z.B. Triangulierung von Polygonen). Weitere Anwendungen mit diesem Abhängigkeitsmuster finden sich in der Computeralgebra (z.B. vollständiges Horner-Schema, Taylor-shift). Die Aufgabe des Planungsalgorithmus besteht darin, die Knoten einer Pyramide der Höhe n auf P Prozessoren abzubilden.

Zunächst werden untere Schranke für die Isoeffizienz von Anwendungen hergeleitet, deren Abhängigkeitsgraph eine Pyramide ist. Unter anderem wird gezeigt, dass die Isoeffizienz P^2 dominiert, wenn alle Knoten die gleiche Bearbeitungszeit haben. Anschließend wird ein Planungsverfahren angegeben (*Pie-Mapping*), das, im Gegensatz zu den bislang bekannten Verfahren, den Graphen in P etwa gleich große, zusammenhängende Zonen teilt und so eine Reduzierung des Kommunikationsaufwandes von einem quadratischen Term zu einem linearen Term erreicht. Dieses macht sich in Experimenten bereits bei einer einfachen ebenenweisen Abarbeitung der Zonen durch eine deutlich höhere Effizienz bemerkbar. Eine weitere Verbesserung der Effizienz wird durch eine blockorientierte Abarbeitungsreihenfolge erreicht. Es wird gezeigt, dass bei dieser Reihenfolge mehr Zeit für die Kommunikation zur Verfügung steht, als bei der ebenenweisen Bearbeitung.

Das in Kapitel 4 betrachtete Lastverteilungsproblem bezieht sich auf unabhängige Aufgaben, die ihrerseits parallel bearbeitet werden können. Das Interesse gilt vor allem dem Fall, in dem weniger Aufgaben als Prozessoren vorhanden sind. Es wird davon ausgegangen, dass sich die Aufgaben gleich verhalten, und dass die Zeit $t(p)$ für die Bearbeitung einer Aufgabe auf p Prozessoren bekannt ist. Das Problem besteht darin, bei gegebener Prozessoranzahl P und Aufgabenanzahl m jeder Aufgabe eine Prozessormenge und einen Startzeitpunkt so zuzuordnen, dass jeder Prozessor nur an einer Aufgabe gleichzeitig arbeitet, und dass die Gesamtberechnungszeit minimiert wird. Im Fall, dass die Aufgaben nicht dieselbe Laufzeitfunktion haben, ist das Problem selbst bei einer konstanten Prozessoranzahl in strengem Sinne NP-hart [**48**]. Im Fall, dass die Aufgaben gleichartig sind, wird ein Planungsalgorithmus angegeben, dessen Laufzeit von $n\,(2n+1)^{P^2-P}\,2^P$ dominiert wird. Diese Schranke kann auf $m\ (k\,t(1))^{P-1}\ 2^P$ reduziert werden,

falls die Bearbeitungszeiten $t(p)$ rational sind. Hierbei ist k das kleinste gemeinsame Vielfache der Zeiten $t(1), \ldots, t(P)$. Da der Algorithmus wegen der exponentiellen Abhängigkeit der Laufzeit von der Prozessorzahl in der Praxis nicht anwendbar ist, wird ein Approximationsalgorithmus angegeben, der sogenannte *phasenparallele Pläne* in einer Zeit berechnet, die von m^2 dominiert wird. Die mit Hilfe dieser Pläne bewirkte Gesamtbearbeitungszeit ist höchstens doppelt so lang wie die eines optimalen Plans.

In Kapitel 5 wird die Platzierung unabhängiger Aufgaben für den Fall betrachtet, dass mehr Aufgaben als Prozessoren vorhanden sind. Es wird von gleichartigen Aufgaben ausgegangen, die am Anfang beliebig auf den Prozessoren verteilt vorliegen. Der Lastverteilungsalgorithmus soll die Aufgaben so umverteilen, dass jeder Prozessor anschließend über dieselbe Aufgabenanzahl verfügt. Dieses Problem ist in der Dissertation Diekmanns [**42**] für datenparallele Anwendungen behandelt worden. In diesen Anwendungen ergeben sich durch die Nachbarschaftsbeziehungen zwischen den Aufgaben kanonisch auch Nachbarschaften zwischen den Prozessoren. Basierend auf diesen Nachbarschaften werden Diffusionstechniken eingesetzt, um eine Balancierung des Systems zu erreichen [**50**]. Diese Techniken können jedoch auch bei unabhängigen Aufgaben eingesetzt werden. Durch die Unabhängigkeit der Aufgaben entfällt in diesem Fall die durch die Anwendung vorgegebene Prozessornachbarschaft. Es ergibt sich somit die Möglichkeit, eine künstliche Prozessortopologie so zu definieren, dass der Lastverteilungsalgorithmus möglichst effizient arbeitet. Wir betrachten zwei auf Diffusion basierende Lastverteilungsalgorithmen, die jeweils in zwei Phasen arbeiten. In der ersten Phase (*Flussberechnungsphase*) wird ein Lastfluss berechnet, und in der zweiten Phase (*Migrationsphase*) wird die Last entsprechend dieses Flusses verteilt.

Der Aufwand dieser Phasen wird experimentell untersucht. Es zeigt sich, dass der Aufwand sowohl von Topologieeigenschaften (wie Grad, Durchmesser, Anzahl der positiven Eigenwerte der Laplacematrix) als auch von der Balanciertheit des Flusses abhängt. Da ein balancierter Fluss zu einer kürzeren Migrationsphase führt, dafür aber aufwendiger in der Berechnung ist, ergibt sich ein Tradeoff zwischen dem Aufwand für die Flussberechnung und dem Aufwand der Migrationsphase.

In Kapitel 6 wird im einzelnen auf das Lastverteilungssystem *VDS* eingegangen. Es wird gezeigt, dass die Konzeption des Systems die im vorangegangenen Abschnitt diskutierten Anforderungen an ein universelles Lastverteilungssystem erfüllt. Ferner wird die Programmierung von VDS Anwendungen anhand eines einfachen Beispiels erläutert und die von VDS unterstützten Lastverteilungstechniken beschrieben. Schließlich zeigt ein experimenteller Vergleich mit anderen Systemen, dass VDS die effizienteste Unterstützung für strikte Baumberechnungen auf Systemen mit verteiltem Speicher bietet.

In Kapitel 7 wird die Anwendung von VDS bei der Parallelisierung der Descartes Methode für reelle Nullstellenisolierung beschrieben. Es zeigt sich, dass diese Anwendung beispielhaft für die Kombination unterschiedlicher Programmierparadigmen (strikte Baumberechnungen und statische Aufgabengraphen) ist, und dass sie den Einsatz eines universellen Lastverteilungssystems erfordert.

Neben der experimentellen Untersuchung der Skalierbarkeit wird die Isoeffizienz der Descartes Methode analysiert. Es wird gezeigt, dass die Isoeffizienz der Methode bei beschränkter Koeffizientenlänge von $P^3 \log^2 P$ dominiert wird.

Die in dieser Arbeit vorgestellten Ergebnisse sind in folgende Publikationen eingegangen.

(1) *Virtual Data Space - A universal load balancing scheme.*
Proceedings of the 4th International Symp. on Parallel Algorithms for Irregularly Structured Problems (IRREGULAR'97), A. Ferreira, R. Lüling und J. Rolim (Editoren), Lecture Notes in Computer Science 1253, Springer Verlag, 1997, S. 159-166.

(2) *Mapping of coarse-grained applications onto workstation-clusters*.
Proceedings of the 5th Euromicro Workshop on Parallel and Distributed Processing, IEEE Computer Society Press, 1997, S. 5–12,
zusammen mit Ralf Diekmann.

(3) *A distributed load balancing algorithm for heterogeneous parallel computing systems.*
Proceedings of the 1998 Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'98), H. R. Arabnia (Editor), CSREA Press, 1998, S. 933-940,
zusammen mit Reinhard Lüling und Stefan Tschöke.

(4) *Parallel real root isolation using the Descartes method.*
Proc. of the 6th International Conference on High Performance Computing, P Banerjee, V.K. Prasanna und B.P Sinha (Editoren), Lecture Notes in Computer Science 1745, Springer Verlag, 1999, S. 261–268,
zusammen mit Werner Krandick.

(5) *A parallel tabu search algorithm for short term lot sizing and scheduling in flexible flow line environments.*
Proceedings of the 16th International Conference on CAD/CAM, Robotics and Factories of the Future, 2000, S. 843-850,
zusammen mit Klaus Brockmann.

(6) *Isoefficiency and the parallel Descartes method.*
In: G. Alefeld, S. Rump, J. Rohn und T. Yamamoto (Editoren), „Symbolic Algebraic Methods and Verification Methods", SpringerMathematics, Springer Verlag, 2001, S. 55–67,
zusammen mit Werner Krandick.

(7) *Towards optimal load balancing topologies.*
Proceedings of the 6th EURO-PAR Conference, A. Bode und T. Ludwig (Editoren), Lecture Notes in Computer Science 1900, Springer Verlag, 2000, S. 277-287.
zusammen mit Burkhard Monien und Robert Preis.

(8) *Virtual Data Space – Load Balancing for Irregular Applications.*
Parallel Computing 26:13-14, 1825-1860, 2000.

KAPITEL 2

Grundlagen

In diesem Abschnitt werden einige grundlegende Definitionen eingeführt. Zunächst wird auf das verwendete Rechenmodell eingegangen. Der Hauptteil dieses Kapitels befasst sich mit dem Konzept der Isoeffizienz, mit dem die Skalierbarkeit von Algorithmen analysiert und klassifiziert werden kann.

2.1. Rechenmodell

Die technisch einfachste Art, parallele Systeme zu konstruieren, besteht darin, Monoprozessorsysteme durch ein Kommunikationsmedium zu verbinden. Die bei einer parallelen Rechnung notwendige Kooperation der Prozessoren wird hierbei durch den Austausch von Nachrichten realisiert. Dem gegenüber besteht die konzeptionell einfachste Art, Algorithmen für parallele Systeme zu entwerfen, darin, von einem gemeinsamen Speicher der Prozessoren auszugehen. Derartige

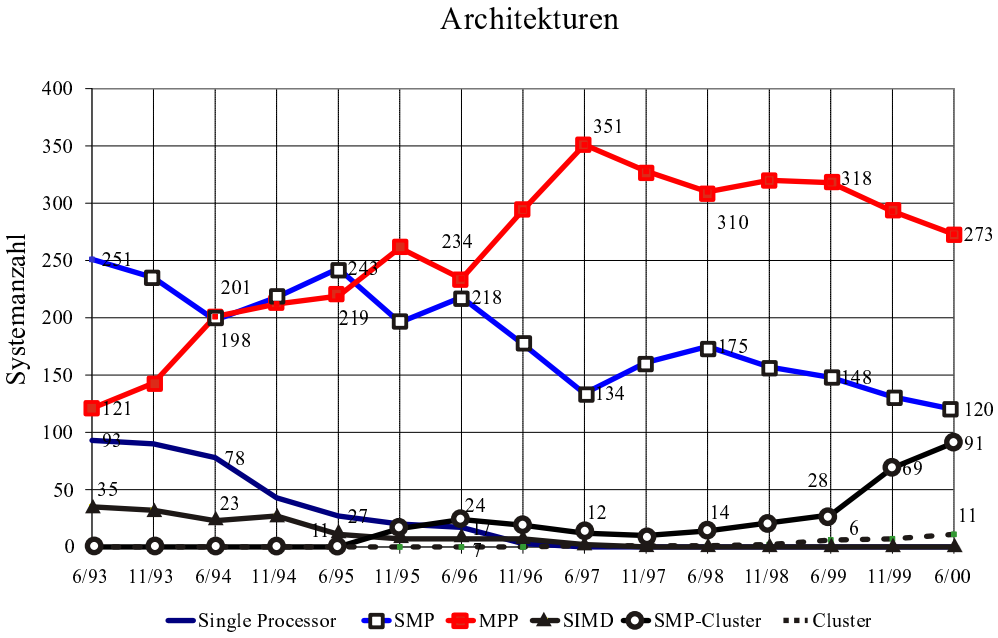


ABBILDUNG 2.1.1. Architekturen der 500 leistungsfähigsten Supercomputer [**46**, Stand 6/2000]. Massiv parallele Systeme (MPP) dominieren die Top-500 Liste. Zunehmend werden Clustersysteme mit SMP-Knoten eingesetzt. Der Anteil reiner Shared-Memory Architekturen ist dagegen seit 1995 rückläufig.

Systeme sind jedoch technisch ungleich schwieriger zu realisieren. Insbesondere ist es zur Zeit nicht möglich, sie in beliebiger Systemgröße herzustellen. Systeme mit 64 Prozessoren können heutzutage schon als "groß" bezeichnet werden. Für den Aufbau größerer Systeme wird daher auf das weniger komfortable Konzept des Nachrichtenaustauschs zurückgegriffen. Abbildung 2.1.1 zeigt, dass etwa drei Viertel der Systeme in der Liste der 500 leistungsfähigsten Supercomputer keinen gemeinsamen Speicher haben.

Die Zielarchitekturen der in dieser Arbeit diskutierten Verfahren sind in erster Linie Systeme, die auf dem Austausch von Nachrichten beruhen. Für Systeme dieser Art sind in der Vergangenheit vielfältige Modellierungen vorgestellt worden [**98**]. Die bekanntesten sind das *LogP*-Modell [**35**] und das *BSP*-Modell [**128**]. Im Gegensatz zu dem LogP-Modell ist jedoch das BSP-Modell kein reines Architekturmodell, da es eine Aufteilung der Anwendung in Kommunikations- und Rechenrunden vorschreibt.

Das LogP-Modell beschreibt ein paralleles System mit Hilfe der Parameter L, o, g für die Beschreibung der Kommunikationseigenschaften und P für die Anzahl der (identischen) Prozessoren. Im einzelnen bezeichnet L die *Latenzzeit*, also die Zeit, die eine Nachricht braucht, um vom Sender zum Empfänger zu gelangen. Der Parameter o bezeichnet den Aufwand der notwendig ist, eine Nachricht zu verschicken bzw. sie zu empfangen (*overhead*). Während dieser Zeit kann ein Prozessor keine anderen Rechnungen durchführen. Der Parameter g (*gap*) gibt die Zeit an, die mindestens zwischen zwei Sende- bzw. Empfangsoperationen vergehen muss. Er modelliert die Kommunikationskapazität des Systems und verhindert, dass das System durch zu viele Nachrichten überlastet wird, und somit die Annahme einer konstanten Latenzzeit nicht mehr realistisch wäre. Wir werden bei der Verwendung des LogP-Modells allerdings davon ausgehen, dass die Kapazität ausreichend groß ist, und $g = o$ annehmen. Ferner werden wir gelegentlich die Latenzzeit von der Größe der Nachricht abhängig machen.

2.2. Skalierbarkeit von Algorithmen

Für die Analyse eines Algorithmus betrachten wir zunächst seine Laufzeit $T(x,\, P)$ für eine Eingabe x im LogP-Modell. Insbesondere ist also $T(x,\, 1)$ die *sequentielle Laufzeit* des Programms. Die Ausführung auf $P > 0$ Prozessoren erzeugt im Allgemeinen zusätzlichen Aufwand, der durch Kommunikation und Wartezeiten entsteht. Die Summe $T_o(x,\, P)$ aller Zeiten, zu denen ein Prozessor nicht mit der Bearbeitung einer Teilaufgabe beschäftigt ist, bezeichnen wir als *Overhead*. Da die Summe der Zeiteinheiten, die jeder einzelne der P Prozessoren mit der Berechnung verbringt durch $P\,T(x,\, P)$ gegeben ist, ergibt sich für den Overhead:

$$T_o(x,\, P) = P\,T(x,\, P) - T(x,\, 1) \;\Rightarrow\; T(x,\, P) = \frac{T(x,\, 1) + T_o(x,\, P)}{P} \quad .$$

Dieses sagt jedoch noch nichts über die Qualität der Parallelisierung aus. Diese wird normalerweise dadurch bewertet, indem man fragt, in welchem Maße ein Programm schneller wird, wenn man die Prozessoranzahl erhöht. Formal verbergen sich hinter dieser Fragestellung die Begriffe des *Speedups* und der *Effizienz*. Der (relative) Speedup $S(x,\, P)$ ist definiert als das Verhältnis zwischen sequentieller und paralleler Laufzeit:

$$S(x,\, P) = \frac{T(x,\, 1)}{T(x,\, P)} = \frac{P\,T(x,\, 1)}{T(x,\, 1) + T_o(x,\, P)} \quad .$$

Die Effizienz $E(x,\ P)$ ist definiert als das Verhältnis zwischen Speedup und Prozessorzahl.

$$\begin{aligned} E(x,\ P) &= \frac{S(x,\ P)}{P} = \frac{T(x,\ 1)}{T(x,\ 1) + T_o(x,\ P)} \\ &= \frac{1}{1 + \frac{T_o(x,P)}{T(x,1)}} \quad . \end{aligned} \tag{2.2.1}$$

Aus Gleichung 2.2.1 wird deutlich, dass die Effizienz von dem Verhältnis des Overheads zur Problemgröße (dargestellt durch die sequentielle Berechnungszeit des Problems) abhängt. Typischerweise steigt der Overhead, wenn die Prozessoranzahl erhöht wird. Umgekehrt verkleinert sich meistens das Verhältnis des Overheads zur sequentiellen Rechenzeit, wenn die Problemgröße wächst. Es ist somit ein immer wiederkehrendes Phänomen, dass die Effizienz sinkt, wenn mehr Prozessoren benutzt werden und dass sie steigt, wenn größere Probleme berechnet werden.

Der Begriff der *Skalierbarkeit eines parallelen Systems* bezieht sich darauf, wie sich die Leistung eines Systems ändert, wenn sich die System- und Problemgrößen ändern. Intuitiv ist ein paralleles System skalierbar, wenn seine Leistungsfähigkeit bei gleichzeitiger Erhöhung der System- und Problemgröße wächst [**114**]. Ein Konzept, diese intuitive Beschreibung zu quantifizieren, ist die von Kumar u.a. eingeführte *Isoeffizienz* [**90, 65**]. Die Isoeffizienz einer parallelen Anwendung gibt an, in welchem Maße die Problemgröße erhöht werden muß, um bei wachsender Prozessorzahl eine vorgegebene Effizienz einzuhalten. Die Idee ist, Gleichung (2.2.1) wie folgt umzuformen:

$$T(x,\ 1) = \underbrace{\frac{E(x,\ P)}{1 - E(x,\ P)}}_{=:\,K} T_o(x,\ P) \quad , \tag{2.2.2}$$

wobei K eine Konstante ist, die von der vorgegeben Effizienz abhängt. Die Isoeffizienz wird in [**65**] wie folgt definiert.

> Through algebraic manipulations, we can use this equation to obtain $T(x,\ 1)$ as a function of P. This function dictates how $T(x,\ 1)$ must grow to maintain a fixed efficiency as P increases. This is the system's *isoefficiency function.*

Demnach kann die Isoeffizienz wie folgt charakterisiert werden. Wenn e eine gegebene Effizienz ist, und $I_e(P)$ die Isoeffizienzfunktion, dann impliziert $T(x,\ 1) = I_e(P)$, dass $E(x,\ P) = e$. Diese Eigenschaft reicht jedoch nicht, um die Isoeffizienzfunktion zu definieren, da es mehrere Eingaben mit derselben sequentiellen Berechnungszeit und unterschiedlichen Effizienzen geben könnte. Ebenso ist es möglich, dass mehrere Eingaben zwar unterschiedliche sequentielle Berechnungszeiten haben, trotzdem aber dieselbe Effizienz erreichen. Beide Situationen entstehen bei parallelen Suchalgorithmen. Der Grund ist, dass die Effizienz nicht nur von der sequentiellen Laufzeit (also von der Anzahl der Suchknoten), sondern auch von der Form des Suchbaumes abhängt.

Um auch in diesen Situationen etwas über die Skalierbarkeit aussagen zu können, fragen wir deshalb nach der kleinsten sequentiellen Rechenzeit, die eine vorgegebene Effizienz garantiert. Falls X die Menge der Eingaben eines parallelen Programms ist, definieren wir

$$I_e(P) := \inf \{T(x,\ 1) \mid (\forall\, y \in X)\, T(y,\ 1) \geq T(x,\ 1) \ \Rightarrow\ E(y,\ P) \geq e\} \quad . \tag{2.2.3}$$

Die Isoeffizienz ist genau dann nicht definiert, wenn die Menge auf der rechten Seite von Gleichung (2.2.3) leer ist. Also genau dann, wenn für jede sequentielle Laufzeit eine Eingabe existiert, deren sequentielle Berechnung mindestens so lange dauert, deren Effizienz jedoch auf

TABELLE 2.2.1. Isoeffizienzfunktionen einiger paralleler Algorithmen.

Algorithmus	Isoeffizienz	Architektur
A*-Suche [**99**]	$\sim P \log^2 P$	Hypercube
Faktorisierung dünn besetzter Matrizzen [**67**]	$\sim P\sqrt{P}$	Clique
Matrix-Vektor Produkt [**141**]	$\sim P\sqrt{P}$	Clique
All-pairs shortest path (Dijkstra) [**91**]	$\preceq (P \log P)^{3/2}$	Hypercube
All-pairs shortest path (Dijkstra) [**91**]	$\preceq P^{9/5}$	Gitter
Descartes Methode ([**39**], Kapitel 7)	$\preceq P^3 \log^2 P$	Clique
FFT [**68**]	$\sim P^{\frac{e}{1-e}\frac{L}{t}} \log P$	Hypercube

P Prozessoren den Wert e nicht erreicht. In diesem Fall ist es sinnvoll, von einer nicht skalierbaren Anwendung zu sprechen. Wenn $I_e(P)$ definiert ist, dann ist garantiert, dass jede Eingabe, deren sequentielle Laufzeit größer als $I_e(P)$ ist, mindestens die Effizienz e auf P Prozessoren erreicht:

$$T(x,\,1) > I_e(P) \text{ impliziert } E(x,\,P) \geq e \quad . \tag{2.2.4}$$

Trotz einer fehlenden rigorosen Definition ist das Konzept der Isoeffizienz in vielen Arbeiten für die Analyse der Skalierbarkeit von Algorithmen herangezogen worden (vgl. Tabelle 2.2.1). In dem folgenden Abschnitt wird das Konzept der Isoeffizienz formal definiert und es werden einige grundlegende Eigenschaften hergeleitet. Abschnitt 2.2.2 setzt die Isoeffizienz mit der von Kruskal u.a. eingeführten Komplexitätsklasse der *effizient parallelen* Algorithmen (*EP*) in Verbindung [**88**]. Schließlich wird in Abschnitt 2.2.3 die Anwendung der Isoeffizienz expemplarisch für *Polyalgorithmen* gezeigt.

2.2.1. Definition und Eigenschaften der Isoeffizienz. Bislang ist die Isoeffizienzfunktion für die Analyse paralleler Algorithmen benutzt worden, bei denen ein direkter Zusammenhang zwischen der sequentiellen Laufzeit und der Effizienz besteht. In [**39**] wurde die erste rigorose Definition der Isoeffizienz eingeführt. Sie liefert auch dann sinnvolle Aussagen, wenn dieser direkte Zusammenhang zwischen sequentieller Laufzeit und Effizienz nicht gegeben ist.

DEFINITION 2.2.1. Es sei ein paralleler Algorithmus gegeben und es sei X die Menge der erlaubten Eingaben. Für jedes $x \in X$ und $P \in \mathbb{N}$ sei $T(x,\,P)$ die parallele Berechnungszeit der Eingabe x auf P Prozessoren. Es sei

$$E(x,\,P) := \frac{T(x,\,1)}{P\,T(x,\,P)}$$

die *Effizienz* dieser Berechnung. Für $e \in [0,\,1)$ sei

$$S_e(P) := \{T(x,\,1) \mid (\forall\, y \in X)\ T(y,\,1) \geq T(x,\,1) \Rightarrow E(y,\,P) \geq e\}$$

die Menge aller sequentiellen Laufzeiten, die die Effizienz e garantieren. Falls $S_e(P)$ nicht leer ist, dann ist die *Isoeffizienzfunktion* $I_e(P)$ definiert durch

$$I_e(P) := \inf\, S_e(P);$$

falls $S_e(P)$ leer ist, dann ist $I_e(P)$ undefiniert.

THEOREM 2.2.2. *Die Isoeffizienzfunktion ist monoton in e. Genauer,*

$$e_1 < e_2 \textit{ impliziert, dass } I_{e_1}(P) \leq I_{e_2}(P) \tag{2.2.5}$$

für alle Effizienzen $e_1,\ e_2$ und jede Prozessoranzahl P.

BEWEIS. Die Behauptung folgt unmittelbar aus $S_{e_2}(P) \subseteq S_{e_1}(P)$. □

Das nächste Theorem zeigt, dass sich eine obere Schranke für die Isoeffizienz aus einer oberen Schranke für die parallele Berechnungszeit herleiten lässt. Es ist klar, dass sich aus einer oberen Schranke für die parallele Berechnungszeit sofort eine untere Schranke für die Effizienz ergibt. Intuitiv führt nun eine Unterschätzung der Effizienz zu einer Überschätzung der Problemgröße, die man benötigt, um eine gegebene Effizienz zu erreichen.

THEOREM 2.2.3. *Die Funktionen $\underline{E} : X \times \mathbb{N} \to (0,\, 1]$ und $\overline{E} : X \times \mathbb{N} \to (0,\, 1]$ seien untere bzw. obere Schranken für die Effizienz E,*

$$\underline{E}(x,\, P) \leq E(x,\, P) \leq \overline{E}(x,\, P)$$

für alle $x \in X$ und $P \in \mathbb{N}$. Für alle Effizienzen $e \in (0,\, 1)$ sei

$$\begin{aligned} \underline{S}_e(P) &:= \{T(x,\, 1) \mid (\forall\, y \in X)\ T(y,\, 1) \geq T(x,\, 1) \Rightarrow \underline{E}(y,\, P) \geq e\} \quad \textit{und} \\ \overline{S}_e(P) &:= \left\{T(x,\, 1) \mid (\forall\, y \in X)\ T(y,\, 1) \geq T(x,\, 1) \Rightarrow \overline{E}(y,\, P) \geq e\right\} \quad . \end{aligned}$$

Weiterhin sei

$$\underline{I}_e(P) := \inf \overline{S}_e(P) \ \text{, und } \ \overline{I}_e(P) := \inf \underline{S}_e(P)$$

wann immer die entsprechenden Mengen nicht leer sind. Dann gilt

$$\underline{I}_e(P) \leq I_e(P) \leq \overline{I}_e(P) \tag{2.2.6}$$

für jede Prozessoranzahl P und alle Effizienzen e.

BEWEIS. Unmittelbar aus $\underline{S}_e(P) \subseteq S_e(P) \subseteq \overline{S}_e(P)$. □

DEFINITION 2.2.4. Eine Menge reeller Zahlen ist *diskret*, falls eine positive untere Schranke für den absoluten Betrag der Differenz je zweier Elemente existiert.

In jedem Rechenmodell, in dem die Zeit für die Basisoperationen Vielfache einer festen Einheitszeit sind, sind die Rechenzeiten diskret.

THEOREM 2.2.5. *Wenn die Berechnungszeiten diskret sind, dann gelten die folgenden Aussagen.*

(1) $I_e(P) \in S_e(P)$, *und*
(2) $T(x,\, 1) \geq I_e(P)$ *impliziert* $E(x,\, P) \geq e$.
(3) $T(x,\, 1) \geq I_e(P)$ *impliziert* $T(x,\, 1) \geq \frac{e}{1-e}\, T_o(x,\, P)$.

BEWEIS. Die erste Aussage ist wahr, da wegen der Diskretheit $\inf S_e(P) = \min S_e(P)$ gilt. Wenn $T(x,\, 1) = I_e(P)$, folgt aus der ersten Aussage $T(x,\, 1) \in S_e(P)$, und somit $E(x,\, P) \geq e$. Falls $T(x,\, 1) > I_e(P)$ folgt die zweite Aussage aus der charakteristischen Eigenschaft (2.2.4) der Isoeffizienzfunktion. Die dritte Aussage ist eine direkte Folgerung aus der zweiten Aussage. □

Üblicherweise sinkt die Effizienz einer Anwendung, wenn die Prozessoranzahl erhöht wird. Die folgende Definition beschreibt diesen Zusammenhang.

DEFINITION 2.2.6. Eine Effizienzfunktion erfüllt die *speed-up* Eigenschaft, wenn für alle $x \in X$ gilt:

$$P_1 < P_2 \ \text{ impliziert } \ E(x,\, P_1) > E(x,\, P_2) \ .$$

THEOREM 2.2.7. *Falls die Effizienzfunktion die speed-up Eigenschaft erfüllt, ist die Isoeffizienzfunktion monoton in P. Es gilt*

$$P_1 < P_2 \text{ impliziert } I_e(P_1) \leq I_e(P_2)$$

für alle Effizienzen e und jede Prozessoranzahl P_1 und P_2.

BEWEIS. Der Beweis erfolgt indirekt. Es seien $P_1 < P_2$ natürliche Zahlen mit $I_e(P_1) > I_e(P_2)$. Dann gilt $S_e(P_2) \not\subset S_e(P_1)$ und folglich gibt es ein $x \in X$ mit $x \in S_e(P_2) \setminus S_e(P_1)$. Folglich gibt es ein $y \in X$ mit $T(y,1) \geq T(x,1)$ und $E(y,P_1) < e \leq E(y, P_2)$. Dieses widerspricht der speed-up Eigenschaft von E. □

Falls die folgende Bedingung zutrifft, ist es möglich, von der Effizienz einer Eingabe auf ihre sequentielle Berechnungszeit zurückzuschließen.

DEFINITION 2.2.8. Eine Effizienzfunktion erfüllt die *speed-up* Eigenschaft, wenn für alle $x,\ y \in X$ und $P > 1$ gilt:

$$T(x,\ 1) < T(y,\ 1) \text{ genau dann wenn } E(x,\ P) < E(y,\ P)\ .$$

THEOREM 2.2.9. *Wenn eine Effizienzfunktion E die size-up Eigenschaft erfüllt, dann gelten die folgenden Aussagen für alle $x \in X$ und $P \in \mathbb{N}$.*

(1) $E(x,\ P) = e$ *impliziert* $T(x,\ 1) = I_e(P)$, *und*
(2) $E(x,\ P) \geq e$ *impliziert* $T(x,\ 1) \geq I_e(P)$.

BEWEIS. Für den Beweis der ersten und eines Teils der zweiten Aussage sei $x \in X$ so gewählt, dass $E(x,\ P) = e$. Wegen der size-up Eigenschaft gilt $T(x,\ 1) \in S_e(P)$. Da alle Eingaben mit einer kleineren sequentiellen Laufzeit eine geringere Effizienz haben, ist $T(x,\ 1)$ das kleinste Element von $S_e(P)$ und somit gilt $T(x,\ 1) = I_e(P)$. Um den Rest der zweiten Aussage zu zeigen, sei $x \in X$ so gewählt, dass $E(x,\ P) > e$. Nun gilt wiederum $T(x,\ 1) \in S_e(P)$ und somit $T(x,\ 1) \geq I_e(P)$. □

Wenn die Berechnungszeiten diskret sind und die Isoeffizienzfunktion die size-up Eigenschaft erfüllt, dann gilt

$$E\,(x, P) \leq e \text{ impliziert } T\,(x, 1) \leq I_e\,(P)\ .$$

BEWEIS. Die Aussage folgt unmittelbar aus den Theoremen 2.2.9(1) und 2.2.5(2). □

Abbildung 2.2.1 zeigt anhand einer parallelen Methode für reelle Nullstellenisolierung, wie Isoeffizienzgraphen experimentell konstruiert werden können. Jeder Graph in dem linken Diagramm wurde mit einer anderen Prozessoranzahl gemessen. Hierbei wurde jeweils die Effizienz verschiedener Eingabepolynome (Mignotte Polynome) der Grade $100,\ 200, \ldots, 500$ gemessen. Die Effizienz jeder Messung ergibt einen Punkt im linken Diagramm. Die Graphen entstehen durch Verbinden der Punkte, die zur gleichen Prozessoranzahl gehören. Das rechte Diagramm zeigt die Isoeffizienzgraphen. Jeder einzelne Graph wird konstruiert, indem man die Graphen im linken Diagramm durch horizontale Linien schneidet. Dabei ergibt jeder Schnittpunkt eine Prozessoranzahl und eine sequentielle Rechenzeit— und somit einen Punkt im rechten Diagramm. In diesem Fall entstehen die Graphen durch Verbinden der Punkte, die zu derselben Effizienz gehören.

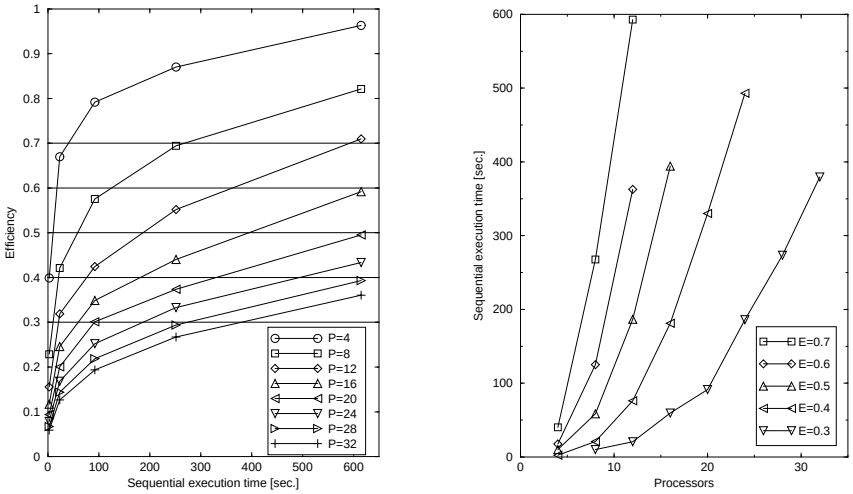


ABBILDUNG 2.2.1. Die Diagramme zeigen, wie Isoeffizienzgraphen experimentell konstruiert werden können. In diesem Beispiel erfüllt die Effizienzfunktion sowohl die speed-up als auch die size-up Bedingung.

2.2.2. Isoeffizienz und die Klasse EP. Kruskal, Rudolph und Snir klassifizieren in [**88**] parallele Algorithmen nach der Arbeit und der parallelen Rechenzeit. Sie betrachten Algorithmen, bei denen die Anzahl der benutzten Prozessoren $p(x)$ von der Eingabe x abhängt (*größenabhängige* Algorithmen). Für das Rechenmodell wird angenommen, dass es *selbstsimulierend* ist. Das heißt, dass die Zeit, die eine Maschine mit p Prozessoren benötigt, um einen Schritt einer Maschine mit $q > p$ Prozessoren zu simulieren, von q/p dominiert wird. Die Simulationszeit ist dagegen konstant, wenn die simulierte Maschine weniger Prozessoren als die simulierende hat. Demnach gilt für die parallele Rechenzeit [**88**, Gleichung 3.1]

$$T(x,\, P) \sim \lceil p(x)/P \rceil \; T\,(x,\, p(x)) \quad . \tag{2.2.7}$$

Ein Algorithmus ist *effizient*, wenn die gesamte Arbeit $T(x,\, p(x)) \cdot p(x)$ von der sequentiellen Laufzeit $T(x,\, 1)$ dominiert wird. Die Klasse der effizienten Algorithmen wird weiter nach der parallelen Rechenzeit $T(x,\, p(x))$ unterteilt.

DEFINITION 2.2.10. *EP* (Efficient, Polynomially fast; oder Efficient Parallel) ist die Klasse der effizienten Algorithmen, deren parallele Rechenzeit für ein $\varepsilon < 1$ die Ungleichung

$$T\,(x,\, p(x)) \leq T(x,\, 1)^{\varepsilon}$$

erfüllt.

Basierend auf dieser Definition werden Problemklassen wie folgt eingeführt.

DEFINITION 2.2.11. Ein Problem ist in der *Klasse EP*, wenn es einen Algorithmus für das Problem gibt, der bezüglich aller sequentiellen Algorithmen für dasselbe Problem in der Klasse EP liegt.

Effiziente Algorithmen stehen in einer engen Beziehung zu Algorithmen, deren Isoeffizienzfunktion für alle $P \in \mathbb{N}$ definiert ist. Analog zu Definition 2.2.11 definieren wir die Problemklasse PI (polynomiell isoeffizient).

DEFINITION 2.2.12. Ein Problem ist in der *Klasse PI*, wenn es einen Algorithmus für das Problem gibt, dessen Isoeffizienzfunktion $I_e(P)$ bezüglich aller sequentieller Algorithmen für dasselbe Problem polynomiell für eine Effizienz e ist.

In diesem Abschnitt werden wir zeigen, dass die Klasse PI eine Unterklasse von EP ist. Ferner zeigen wir, dass es Algorithmen gibt, die zwar in EP sind, trotzdem aber eine exponentielle Isoeffizienzfunktion haben. Wir benutzen hierfür eine äquivalente Charakterisierung der Klasse EP [**88**, Theoreme 3.1 (3) und 3.2 (3)].

THEOREM 2.2.13. *Es sei k eine Konstante. Ein Problem kann genau dann durch einen (größenabhängigen) Algorithmus der Klasse EP berechnet werden, wenn es durch einen (größenunabhängigen) Algorithmus, für dessen parallele Laufzeit*

$$T(x,\, P) \preceq T(x,\, 1)/P + P^k$$

gilt, berechnet werden kann.

Das folgende Lemma stellt eine Verbingung zwischen EP und dem Isoeffizienzkonzept her.

LEMMA 2.2.14. *Es sei ein paralleler Algorithmus mit der Rechenzeit $T(x,\, P)$ gegeben. Falls $T(x,\, P) \preceq T(x,\, 1)$, dann sind folgende Aussagen äquivalent.*

(1) *Es gibt eine Konstante k so dass*

$$T(x,\, P) \preceq T(x,\, 1)/P + P^k \ .$$

(2) *Es gibt eine Konstante k und eine Effizienz e so dass für alle $x \in X$*

$$T(x,\, 1) \geq P^k \ \Rightarrow\ E(x,\, P) \geq e \ .$$

BEWEIS. ("1. $\Rightarrow$ 2.") Es sei k eine Konstante mit $T(x,\, P) \preceq T(x,\, 1)/P + P^k$. Es sei c so gewählt, dass $T(x,\, P) \leq c\,(T(x,\, 1)/P + P^k)$ und es sei $e := 1/(2\,c)$. Betrachten wir nun eine Eingabe x mit $T(x,\, 1) \geq P^{k+1}$. Es folgt

$$\begin{aligned} T(x,\, P) &\leq c\,\frac{T(x,\, 1) + P^{(k+1)}}{P} \leq 2c\,\frac{T(x,\, 1)}{P} \\ \Rightarrow\ E(x,\, P) = \frac{T(x,\, 1)}{P\,T(x,\, P)} &\geq \frac{1}{2c} = e \ . \end{aligned}$$

("2. $\Rightarrow$ 1.") Nun sei k und e so gewählt, dass für alle Eingaben x mit $T(x,\, 1) \geq P^k$ die Aussage $E(x,\, P) \geq e$ gilt. Es sei x eine Eingabe. Falls $T(x,\, 1) \geq P^k$ erhalten wir

$$\begin{aligned} E(x,\, P) &\geq e \\ \frac{T(x,\, 1)}{P\,T(x,\, P)} &\geq e \\ \Rightarrow \frac{1}{e}\frac{T(x,\, 1)}{P} &\geq T(x,\, P) \\ \Rightarrow T(x,\, P) &\preceq \frac{T(x,\, 1)}{P} \preceq \frac{T(x,\, 1)}{P} + P^k \ . \end{aligned}$$

Andernfalls, falls $T(x,\, 1) < P^k$, gilt

$$T(x,\, P) \preceq T(x,\, 1) < P^k \preceq \frac{T(x,\, 1)}{P} + P^k \ .$$

□

THEOREM 2.2.15. *Es sei ein paralleler Algorithmus gegeben, dessen parallele Rechenzeit von seiner sequentiellen dominiert wird. Der Algorithmus ist in EP, wenn seine Isoeffizienzfunktion $I_e(P)$ polynomiell ist.*

BEWEIS. Es sei c eine Konstante mit $I_e(P) \leq c\,P^k$. Gemäß der charakteristischen Eigenschaft der Isoeffizienzfunktion gilt

$$T(x,\,1) > c\,P^k \;\Rightarrow\; E(x,\,P) \geq e \;\;.$$

Somit gibt es eine Konstante K, so dass

$$T(x,\,1) > P^K \;\Rightarrow\; E(x,\,P) \geq e \;\;.$$

□

THEOREM 2.2.16. *Die Problemklasse PI ist eine Unterklasse von EP.*

Algorithmus 1 Durch die gleichzeitige Simulation der parallelen und der sequentiellen Berechnung erhalten wir einen Algorithmus, dessen sequentielle Rechenzeit stets größer die parallele ist.

W (Algorithmus A, Eingabe x, $P \in \mathbb{N}$)

if $P \leq 2$ **then**
  Führe A auf Prozessor 1 mit Eingabe x aus.
else
  parbegin
  Führe A auf Prozessor 1 mit Eingabe x aus.
  Führe A auf den Prozessoren $2, \ldots, P$ mit Eingabe x aus.
  Halte sobald eine der beiden Rechnungen terminiert.
  parend

BEWEIS. Es sei A ein Algorithmus und $I_e^{(A)}(P)$ sei seine Isoeffizienzfunktion. Die parallele Rechenzeit $T^{(A)}(x,\,1)$ wird nicht notwendigerweise von der sequentiellen Laufzeit dominiert. Wir betrachten daher den Algorithmus 1. Bei jeder Eingabe x berechnet $W(A,\,x,\,P)$ dieselbe Ausgabe wie A. Die parallele Rechenzeit $T^{(W)}(A,\,x,\,P)$ von W wird wegen

$$T^{(W)}(A,\,x,\,P) = \min\left(T^{(A)}(x,\,1),\,T^{(A)}(x,\,P-1)\right) \leq T^{(A)}(x,\,1) = T^{(W)}(A,\,x,\,1)$$

von der sequentiellen Rechenzeit dominiert. Für die Effizienz von W erhalten wir

$$\begin{aligned} E^{(W)}(A,\,x,\,P) &= \max\left(\frac{1}{P},\,\frac{P-1}{P}\,E^{(A)}(x,\,P-1)\right) \geq \frac{P-1}{P}\,E^{(A)}(x,\,P-1) \\ &\geq \frac{1}{2}\,E^{(A)}(x,\,P-1) \;\;. \end{aligned}$$

Mittels dieser Ungleichung können wir die Mengen $S_e^{(A)}(P)$ und $S_e^{(W)}(P)$ in Beziehung setzen.

$$\begin{aligned} S_e^{(W)}(P) &= \left\{ T(x,\,1) \mid (\forall\, y \in X)\, T(y,\,1) \geq T(x,\,1) \;\Rightarrow\; E^{(W)}(A,\,y,\,P) \geq e \right\} \\ &\supseteq \{ T(x,\,1) \mid (\forall\, y \in X)\, T(y,\,1) \geq T(x,\,1) \\ &\qquad\qquad \Rightarrow\; \frac{P-1}{P}\, E^{(A)}(y,\,P-1) \geq e \Big\} \\ &= S^{(A)}_{e\frac{P}{P-1}}(P-1) \\ &\supseteq S^{(A)}_{2e}(P-1)\ . \end{aligned}$$

Folglich gilt $I_e^{(W)}(P) \leq I_{2e}^{(A)}(P-1)$.

Es sei nun Υ ein Problem aus PI und es sei A ein Algorithmus, dessen Isoeffizienzfunktion $I_e^{(A)}(P)$ polynomiell in P für eine Effizienz e ist. Dann ist die Isoeffizienzfunktion $I_{e/2}^{(W)}(P)$ ebenso polynomiell. Wegen Theorem 2.2.15 ist somit der Algorithmus $W(A,\,x,\,P)$ in EP, und das Problem Υ ist in der Problemklasse EP. □

THEOREM 2.2.17. *Es gibt einen Algorithmus in EP, dessen Isoeffizienzfunktion $I_{1/2}(P)$ nicht durch eine polynomielle Funktion in P beschränkt werden kann.*

Algorithmus 2 Ein paralleler Algorithmus, der sehr ineffizient wird, wenn die sequentielle Laufzeit kleiner als P ist. Der Parameter p gibt den Prozessorindex an. Die Funktion $f(n)$ ist rekursiv durch $f(1) = 4$ und $f(n) = 2^{f(n-1)}$ definiert.

---

bad_algo ($n \in \mathbb{N},\, P \in \mathbb{N},\, p \in \{0, \ldots, P-1\}$)

if $f(n) \leq P$ **then**
  if $p = 0$ **then** führe $f(n)$ Operationen aus.
  else terminiere.
else
  if $p < f(n) \mod P$ **then** führe $\lceil f(n)/P \rceil$ Operationen aus.
  else führe $\lfloor f(n)/P \rfloor$ Operationen aus.

BEWEIS. Für den Beweis betrachten wir den Algorithmus 2. Die sequentielle Rechenzeit von bad_algo$(n,\,1,\,0)$ ist gegeben durch $T(n,\,1) = f(n) \geq 4$. Wenn der Algorithmus auf P Prozessoren mit Eingabe n ausgeführt wird, ist seine gesamte Arbeit gleich der sequentiellen Rechenzeit $T(n,\,1)$. Falls $T(n,\,1) > P$, ist die Effizienz mindestens $1/2$, da

$$E(n,\,P) = \frac{f(n)}{P\,\left\lceil \frac{f(n)}{P} \right\rceil} \geq \frac{f(n)}{P\,\left(\frac{f(n)}{P} + 1\right)} = \frac{f(n)}{f(n)+P} \geq \frac{f(n)}{2\,f(n)} = \frac{1}{2}\ .$$

Der Algorithmus bad_algo ist also in EP. Abbildung 2.2.2 zeigt die Beziehung zwischen der sequentiellen Laufzeit und der Effizienz.

Es sei nun $m,\, P \in \mathbb{N}$ mit $f(m) = P$. Da die sequentiellen Rechenzeiten diskret sind, gibt es $n \in \mathbb{N}$ mit $T(n,\,1) = I_{1/2}(P)$ und $E(n,\,P) \geq 1/2$. Dieses impliziert bereits, dass $T(n,\,1) > P$. Falls nämlich $T(n,\,1) \leq P$ wäre, dann wäre die Effizienz des Algorithmus

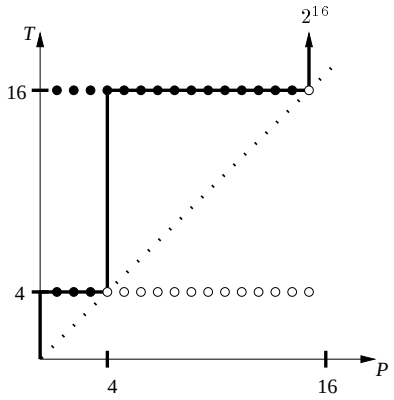


ABBILDUNG 2.2.2. Die Menge der sequentiellen Rechenzeiten ist gegeben durch $\{f(n) \mid n \in \mathbb{N}\}$. Für jede sequentielle Zeit T und jede Prozessoranzahl P gibt es eine Berechnung. Ihre Effizienz ist $\geq 1/2$ genau dann wenn $T > P$ (die gefüllten Punkte). Die Linie zeigt den Graph von $I_{1/2}(P)$.

$E(n,\, P) = 1/P = 1/f(m) \leq 1/4$. Die kleinste sequentielle Rechenzeit, die größer als P ist, ist $f(m+1) = 2^{f(m)}$. Da $E(m+1,\, P) \geq 1/2$, erhalten wir $I_{1/2}(P) = 2^P$. Da es unendlich viele Werte für P mit $I_{1/2}(P) = 2^P$ gibt, kann die Isoeffizienzfunktion $I_{1/2}(P)$ nicht durch ein Polynom in P beschränkt werden. □

2.2.3. Isoeffizienz von Polyalgorithmen . Zum Schluss dieses Kapitels leiten wir ein Resultat her, das bei der Analyse der Isoeffizienz der Descartes Methode benötigt wird. Es ist gleichzeitig ein gutes Beispiel für die Verwendung des Isoeffizienzkonzepts. Und zwar betrachten wir die Isoeffizienzfunktion von Algorithmen, die auf unterschiedliche Arten parallelisiert werden können. Wenn es möglich ist, die Effizienz der verschiedenen Parallelisierungen für eine gegebene Eingabe vorauszusagen, wird man die Parallelisierung wählen, die die größte Effizienz verspricht. Das folgende Theorem analysiert die Isoeffizienz des resultierenden Polyalgorithmus.

THEOREM 2.2.18. *Es seien A und B parallele Algorithmen mit derselben Eingabemenge X und derselben sequentiellen Rechenzeit $T(x,\, 1)$ für jede Eingabe x. Es seien $E^{(A)}$ und $E^{(B)}$ die Effizienzfunktionen der Algorithmen A und B, und es seien $I_e^{(A)}$ und $I_e^{(B)}$ die Isoeffizienzfunktionen. Der Polyalgorithmus C verwende bei Eingabe x den Algorithmus A, falls $E^{(A)}(x,\, P) > E^{(B)}(x,\, P)$ und sonst Algorithmus B. Es sei $E^{(C)}$ die Effizienzfunktion und $I_e^{(C)}$ die Isoeffizienzfunktion von Algorithmus C. Dann gelten die folgenden Aussagen.*

(1) $I_e^{(C)}(P) \leq \min\left(I_e^{(A)}\,(P)\,,\, I_e^{(B)}\,(P)\right)$.
(2) Wenn $E^{(A)}$ und $E^{(B)}$ die speed-up Eigenschaft erfüllen, dann wird sie auch von $E^{(C)}$ erfüllt.
(3) Wenn $E^{(A)}$ und $E^{(B)}$ die size-up Eigenschaft erfüllen, dann wird sie auch von $E^{(C)}$ erfüllt. Wenn zusätzlich die Berechnungszeiten diskret sind, gilt $I_e^{(C)}(P) = \min\left(I_e^{(A)}\,(P)\,,\, I_e^{(B)}\,(P)\right)$.

BEWEIS. Die erste Aussage gilt, da $S_e^{(C)}(P) \supseteq S_e^{(A)}(P) \cup S_e^{(B)}(P)$ impliziert, dass

$$\begin{aligned} I_e^{(C)}(P) &= \inf\, S_e^{(C)}(P) \\ &\leq \inf\left(S_e^{(A)}(P) \cup S_e^{(B)}(P)\right) \\ &= \min\left(\inf\, S_e^{(A)}(P),\, \inf\, S_e^{(B)}(P)\right) \\ &= \min\left(I_e^{(A)}(P),\, I_e^{(B)}(P)\right) \quad . \end{aligned}$$

Die zweite Aussage folgt aus der Definition 2.2.6. Um die dritte Aussage zu zeigen, betrachten wir zwei Eingaben x und y mit $T(x,\,1) < T(y,\,1)$. Falls C den Algorithmus A auf x anwendet und den Algorithmus B auf y, dann gilt $E^{(C)}(x, P) = E^{(A)}(x,\, P) < E^{(A)}(y,\, P) \leq E^{(B)}(y,\, P) = E^{(C)}(y,\, P)$; für die anderen Fälle gelten ähnliche Beziehungen. Ebenso impliziert $E^{(C)}(x,\, P) < E^{(C)}(y,\, P)$, dass $T(x,\, 1) < T(y,\, 1)$. Um den zweiten Teil der Aussage zu zeigen, sei $T(x, 1) \in S_e^{(C)}(P)$. Wenn der Algorithmus A auf x angewendet wird, erhalten wir $E^{(A)}(x, P) = E^{(C)}(x,\, P) \geq e$ und somit wegen Theorem 2.2.9(2), $T(x,\, 1) \geq I_e^{(A)}(P)$. Weiterhin folgt wegen Theorem 2.2.5, dass $T(x,\, 1) \in S_e^{(A)}(P)$. Wenn Algorithmus B für x ausgewählt wird, erhalten wir stattdessen $T(x,\, 1) \in S_e^{(B)}(P)$. Es gilt somit $S_e^{(C)}(P) \subseteq S_e^{(A)}(P) \cup S_e^{(B)}(P)$, und folglich $I_e^{(C)}(P) \geq \min\left(I_e^{(A)},\, I_e^{(B)}\right)$. □

KAPITEL 3

Planungsverfahren für die Pyramide

Wenn der Aufgabengraph einer parallelen Anwendung bereits vor der eigentlichen Ausführung bekannt ist, kann die Lastverteilung ebenfalls im voraus festgelegt werden (Ablaufplanung). Wir werden in diesem Fall von *Berechnungen eines Aufgabengraphen* sprechen. Die Aufgabe des Planungsalgorithmus besteht darin, für jede Aufgabe mindestens einen Prozessor und einen Bearbeitungszeitpunkt zu bestimmen, so dass die Datenabhängigkeiten berücksichtigt werden und die Berechnungszeit des Graphen (Makespan) $C_{\max}$ minimiert wird.

Für allgemeine Aufgabengraphen handelt es sich in den meisten Fällen um NP-vollständige Planungsprobleme. Bestimmte Einschränkungen der Graphstruktur oder des Verhältnisses zwischen Aufgabenberechnungszeit und der Kommunikationszeit ermöglichen jedoch polynomielle Algorithmen. In diesem Kapitel betrachten wir Aufgabengraphen, die gitterförmige Abhängigkeiten aufweisen. Ein Knoten $(i,\, j)$ ist somit von den Knoten $(i,\, j-1)$ und $(i-1,\, j)$ abhängig (siehe Abbildung 3.1.1). Anwendungen dieser Graphen finden sich vor allem bei Algorithmen, die auf dynamischer Programmierung beruhen [**62**].

Beispielsweise treten umgekehrte Pyramiden bei der Berechnung einer optimalen Klammerung eines Matrizenprodukts und beim Finden einer optimalen Triangulierung eines konvexen Polygons auf [**32, 12**]. Das Problem eine längste gemeinsame Teilsequenz zweier Zeichenketten zu finden (LCSS-Problem) und darauf aufbauende Probleme aus der Biologie [**92**] erfordern die Berechnung eines Diamantengraphen. Anwendungen der Pyramide finden sich unter anderem in der Computeralgebra (vollständiges Horner-Schema [**93**], Taylor-Shift [**85**]).

Abschnitt 3.1 fasst den Stand der Forschung zusammen. In Abschnitt 3.2 betrachten wir untere Schranken für die Isoeffizienz von Berechnungen der Pyramide. In Abschnitt 3.3 werden zwei bekannte Planungsverfahren analysiert und ein neues Planungsverfahren vorgestellt. Abschnitt 3.4 diskutiert offene Fragen.

3.1. Stand der Forschung

Es sei $G = (V,\, D)$ ein Aufgabengraph und $t(i)$ die Bearbeitungszeit einer Aufgabe $i \in V$. Abweichend vom LogP-Modell wird der Kommunikationsaufwand häufig durch Angabe der Latenzzeit $c(i,\, j)$ modelliert, die entsteht, wenn Daten über die Kante $(i,\, j) \in D$ geschickt werden. Wenn also die Aufgaben i und j auf unterschiedlichen Prozessoren bearbeitet werden und Aufgabe i zum Zeitpunkt t abgearbeitet ist, ist ihr Ergebnis erst zum Zeitpunkt $t + c(i,\, j)$ für die Bearbeitung von j verfügbar. Der Overhead wird bei dieser Modellierung nicht betrachtet. Wir bezeichnen mit $t_{\min}$ die minimale Bearbeitungszeit eines Knotens und mit $c_{\max}$ die maximale Kommunikationszeit. Der Quotient $c_{\max}/t_{\min}$ ist somit ein Maß für die Granularität einer Anwendung.

In der Literatur werden die folgenden Einschränkungen bzgl. der Latenzzeiten betrachtet.

DS: Es sind beliebige Latenzzeiten zulässig (engl. *delay scheduling*).

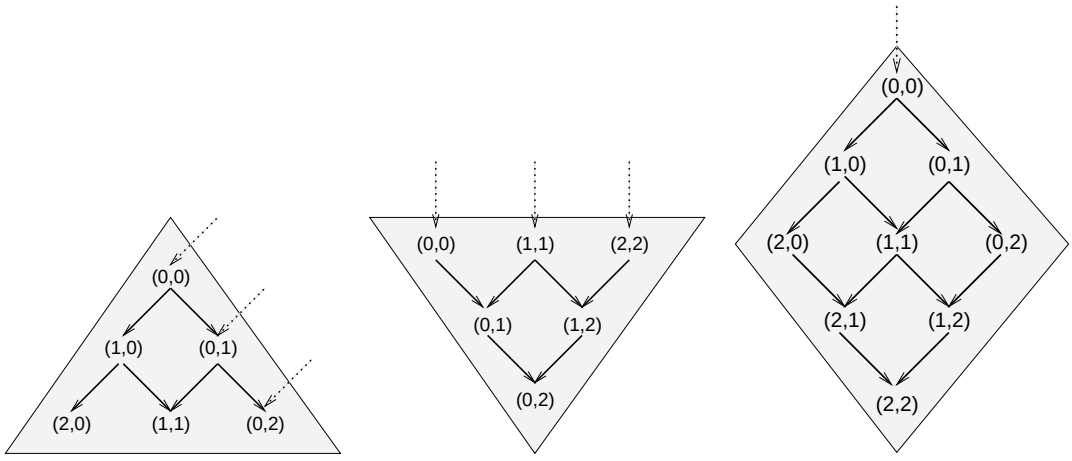


ABBILDUNG 3.1.1. Aufgabengraphen mit gitterförmigen Datenabhängigkeiten.

UDS: Die Latenzzeiten sind gleich einem Wert, der von dem Graphen abhängen kann (engl. *uniform delay scheduling*).

CDS: Die Latenzzeiten sind gleich einer Konstante, die nicht vom Graphen abhängt (engl. *constant delay scheduling*).

SCT: Die Latenzzeiten erfüllen die Bedingung $c_{\max}/t_{\min} < 1$ (engl. *small communication time*).

Wir unterscheiden die Algorithmen danach, ob sie eine Duplizierung von Aufgaben für die Minimierung der Kommunikationskosten durchführen. Abbildung 3.1.2 zeigt eine Zusammenfassung einiger komplexitätstheoretischer Ergebnisse, auf die im Folgenden weiter eingegangen wird.

3.1.1. Planungsverfahren ohne Aufgabenduplizierung. Betrachten wir zunächst die Komplexität des Planungsproblems. Es ist bereits für sehr einfache Graphen NP-vollständig, selbst wenn eine unbeschränkte Prozessoranzahl zur Verfügung steht. So ist das Planungsproblem bereits für Bäume [**23**] und für Bäume mit von den Blättern zur Wurzel gerichteten Kanten (*Rückwärtsbäume*) NP-vollständig [**22**]. Bäume mit Tiefe 1 lassen sich allerdings in polynomieller Zeit planen.

Auch Beschränkungen der Berechnungs- und Kommunikationszeiten beeinflussen die Komplexität. So zeigt Picouleau [**107**], dass das Planungsproblem NP-vollständig ist, wenn $t(i) = c(i, j) = 1$ für alle $i, j \in V$. Einfacher wird das Problem jedoch bei den folgenden Aufgabengraphen, wenn die SCP-Bedingung erfüllt ist. Crétienne zeigt, dass es in dem Fall in linearer Zeit möglich ist, Pläne für Bäume und Rückwärtsbäume zu berechnen [**21**]. Ebenfalls polynomiell planbar sind gerichtete bipartite Aufgabengraphen, wie sie durch das Farming Paradigma generiert werden [**24**].

Eine weitere in polynomieller Zeit planbare Graphklasse sind die *seriell-parallelen* (*SP*) Graphen [**129**]. Sie werden von Anwendungen erzeugt, die dem Fork-Join Paradigma folgen und sind rekursiv wie folgt definiert. Der kleinste SP-Graph besteht aus einem einzelnen Knoten, der gleichzeitig Eingangs- und Ausgangsknoten des Graphen ist. Größere Graphen werden mit den

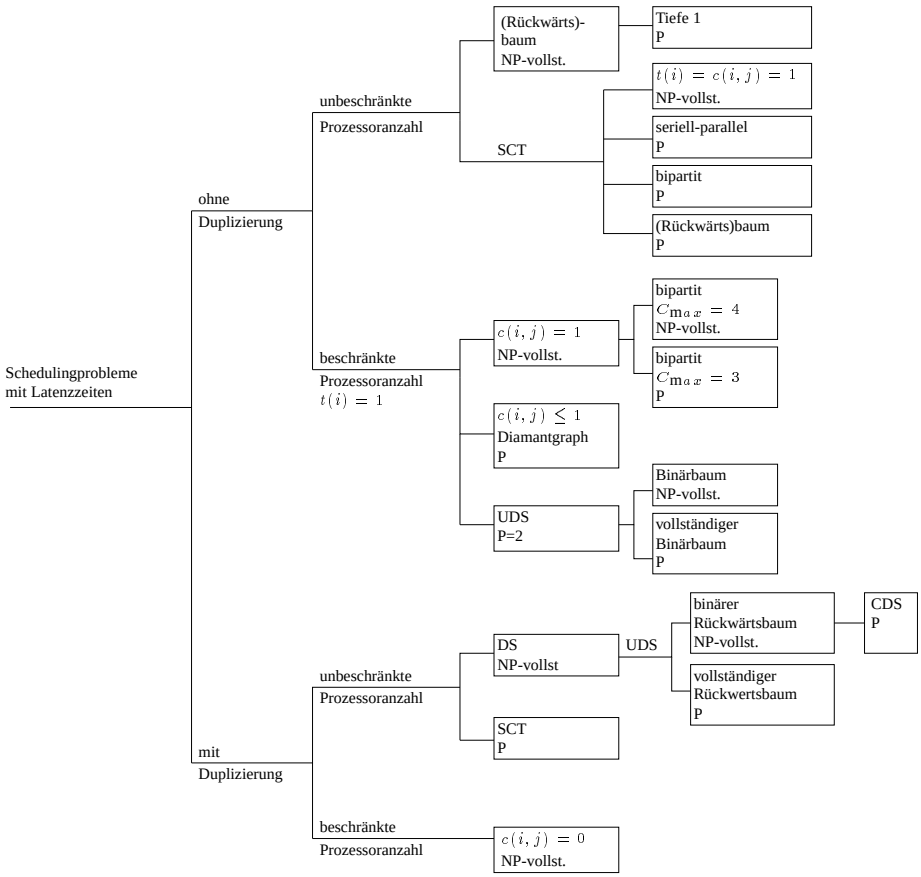


ABBILDUNG 3.1.2. Komplexität von Planungsproblemen mit Latenzzeiten.

Konstruktionsfunktionen SER und PAR gebildet. Seien also $G_1, \ldots G_d$ seriell-parallele Graphen. Dann entsteht SER$(G_1, \ldots G_d)$ durch Hintereinanderschalten der einzelnen Graphen. Beim Hintereinanderschalten wird jeweils der Ausgangsknoten des Vorgängers mit dem Eingangsknoten des Nachfolgers verbunden. Der Eingangsknoten von SER$(G_1, \ldots G_d)$ ist der Eingangsknoten von G_1 und sein Ausgangsknoten ist der Ausgangsknoten von G_d. PAR$(G_1, \ldots G_d)$ verbindet die Eingangsknoten von $G_1, \ldots G_d$ mit einem neuen Eingangsknoten und die Ausgangsknoten mit einem neuen Ausgangsknoten. Die einzelnen Graphen können somit parallel berechnet werden.

Der erste Approximationsalgorithmus für das Planungsproblem mit unbegrenzter Prozessoranzahl wurde von Sarkar vorgestellt [**115**]. Darauf basierend wurde von Gerasoulis und Young der DSC Algorithmus entwickelt und analysiert [**140**]. DSC findet einen Plan für einen Graphen mit n Knoten und m Kanten in Zeit $O\left((n+m)\log n\right)$ und erreicht einen Makespan $C_{\max}$ mit $C_{\max} \leq C^*_{\max}\left(1 + 1/g(G)\right)$. Hierbei ist $C^*_{\max}$ der Makespan eines optimalen Schedules und $g(G)$

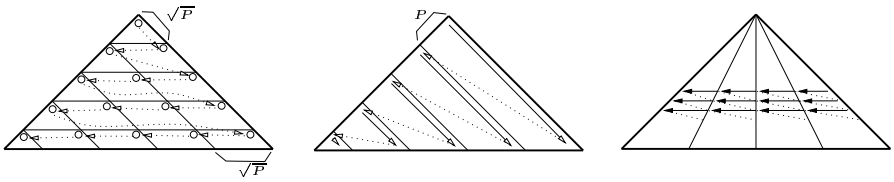


ABBILDUNG 3.1.3. Drei Planungsalgorithmen für die Pyramide.

die Granularität von G, die in diesem Fall wie folgt definiert ist. Für jeden Knoten $j \in V$ sei

$$\begin{aligned} u_j &:= \min_{(i,j)\in D} \{t(i)\} \,/\, \max_{(i,j)\in D} \{c(i,\, j)\} \quad , \\ v_j &:= \min_{(j,\,k)\in D} \{t(k)\} \,/\, \max_{(j,\,k)\in D} \{c(j,\, k)\} \quad \text{und} \\ g_j &:= \min\{u_j,\, v_j\} \quad . \end{aligned}$$

Dann ist $g(G) := \min_{j\in V} \{g_j\}$. Folglich ergibt sich für SCT-Graphen ein Approximationsfaktor von 2.

In realistischen Anwendungen steht jedoch nur eine begrenzte Prozessoranzahl zur Verfügung. Es wurde von Rayward-Smith gezeigt, dass ein NP-vollständiges Planungsproblem vorliegt, wenn $t(i) = c(i,\, j) = 1$ für alle $i,\, j \in V$ [**112**]. Ferner wurde von Hoogeveen u.a. gezeigt, dass es in diesem Fall bereits NP-vollständig ist, zu entscheiden, ob es für einen gerichteten bipartiten Graphen einen Plan mit $C_{\max} \leq 4$ gibt [**72**]. Dieselbe Frage kann für $C_{\max} \leq 3$ in polynomieller Zeit beantwortet werden [**24**]. Afrati u.a. untersuchen die Planung von Binärbäumen auf 2 Prozessoren für den UDS-Fall und zeigen, dass sie im Allgemeinen NP-vollständig ist und für vollständige Binärbäume in polynomieller Zeit durchführbar ist.

Auch für den Fall, dass die Prozessoranzahl begrenzt ist, wurden Heuristiken und Approximationsalgorithmen vorgestellt. Die meisten basieren auf einem *listenbasierten* Ansatz (engl. *list scheduling*). Ein listenbasierter Planungsalgorithmus führt wiederholt folgende Schritte aus. (1) Alle ausführbaren Aufgaben werden gemäß einer geeignet gewählten Priorität in eine Warteschlange eingefügt. Eine Aufgabe ist ausführbar, sobald alle Vorgängeraufgaben abgearbeitet sind. (2) Für die erste Aufgabe in der Warteschlange wird ein geeigneter Prozessor bestimmt. Normalerweise ist dieses ein Prozessor, der die Aufgabe am frühestens berechnen kann.

Einer der effizientesten Planungsalgorithmen ist der listenbasierte ETF-Algorithmus (engl. earliest task first) [**74**]. ETF gewährleistet für den Makespan

$$C_{\max} \leq \left(2 - \frac{1}{P}\right) \widetilde{C}^*_{\max} + \Lambda_G \quad ,$$

wobei $\widetilde{C}^*_{\max}$ der minimale Makespan unter Vernachlässigung der Kommunikationskosten und Λ_G die Länge des kritischen Pfades ist. Boeres u.a. zeigen, dass der ETF-Algorithmus den Diamanten in diagonal von oben links nach unten rechts verlaufende Streifen einteilt, die jeweils den Prozessoren zugewiesen werden. Dieses entspricht dem *Pipelining*-Ansatz (b) in Abbildung 3.1.3. Die Autoren zeigen ferner, dass diese Aufteilung optimal ist, wenn alle Knoten die gleiche Berechnungszeit t haben und die Kommunikationszeiten höchstens so groß wie t sind [**11**].

Lewandowsky u.a. vergleichen den Pipelining-Ansatz mit dem *Diagonalverfahren*, bei dem der Graph in waagerechte Streifen eingeteilt wird [**94**]. Jeder Prozessor bearbeitet etwa $1/P$ der Einträge jedes Streifens (vgl. Abbildung 3.1.3(c)). Nach jedem Streifen synchronisieren sich

die Prozessoren. Die Autoren analysieren beide Ansätze für Systeme mit gemeinsamen Speicher unter der Annahme von exponentiell verteilten Bearbeitungszeiten. Sie kommen zu dem Ergebnis, dass wegen Unbalanciertheiten in jeder Synchronisationsphase das Diagonalverfahren schlechter als der Pipelining-Ansatz ist.

Karypis und Kumar stellen für die umgekehrte Pyramide das zyklische Schachbrettmapping vor [**82**]. Es teilt den Graphen in Zonen mit jeweils P Knoten auf und weist jeweils jedem Prozessor einen Knoten aus jeder Zone zu (Abbildung 3.1.3(a)). Die Autoren zeigen am Beispiel eines auf dynamischer Programmierung beruhenden Algorithmus für die Bestimmung einer optimalen Klammerung eines Matrixprodukts, dass durch Vergrößerung der Eingabe jede Effizienz erreicht werden kann. Kumar u.a. kommen allerdings in [**89**] zu dem falschen Ergebnis, dass die Isoeffizienzfunktion dieser Anwendung von $P^{1.5}$ dominiert wird (Abschnitt 12.4.2). In Abschnitt 3.2 zeigen wir, dass sie in diesem Fall sogar P^3 dominiert.

In Abschnitt 3.3.1 stellen wir das *Pie-Mapping* Verfahren vor, das mit dem Diagonalverfahren vergleichbar ist, jedoch keine globale Synchronisationen benötigt. Ferner werden die ersten beiden in Abbildung 3.1.3 dargestellten Ansätze genauer analysiert und mit dem Pie-Mapping Verfahren verglichen.

3.1.2. Planungsverfahren mit Aufgabenduplizierung. Betrachten wir wieder zunächst die Komplexität für den Fall einer unbegrenzten Prozessoranzahl. Für allgemeine Graphen zeigen Papadimitriou und Yannakakis, dass bereits im UDS-Fall das Planungsproblem NP-vollständig ist [**106**]. Ferner geben sie einen Approximationsalgorithmus mit Approximationsfaktor 2 an. Colin und Crétienne zeigen, dass das Problem im SCT-Fall allerdings in P liegt [**25**]. Ferner geben Jung u.a. für konstante Latenzzeiten (CDS) einen Polynomialzeitalgorithmus an [**80**]. Jakoby und Reischuk zeigen, dass das Planungsproblem im Fall CDS P-vollständig ist [**76**]. Dieselben Autoren zeigen in [**75**], dass es im UDS-Fall für Binärbäume NP-vollständig ist, für vollständige Bäume allerdings noch in P liegt. Das Planungsproblem ist jedoch auch für vollständige Bäume NP-vollständig, wenn beliebige Latenzzeiten zulässig sind [**75**].

Papadimitriou und Ullman betrachten die Planung des $n \times n$-Diamanten für den CDS-Fall [**105**]. Sie zeigen für die parallele Berechnungszeit T und die Anzahl der Kommunikationsvorgänge C, dass für jeden Plan die Beziehung $(C+n)\,T \succeq n^3$ gilt. Weiterhin zeigen sie für die Anzahl D der Kommunikationsoperationen auf dem kritischen Pfad die Beziehung $(D+1)\,T \succeq n^2$.

Die Beschränkung der Prozessoranzahl führt dazu, dass das Planungsproblem bereits dann NP-vollständig ist, wenn alle Latenzzeiten 0 sind [**61**]. Es gibt daher eine Reihe von Arbeiten, in denen heuristische Planungsalgorithmen (für beliebige Latenzzeiten) vorgestellt werden. Park u.a. stellen in [**57**] die listenbasierte Planungsheuristik DFRN (Duplication First Reduction Next) vor. Ein experimenteller Vergleich mit anderen, teilweise ohne Duplikation arbeitenden Verfahren, zeigt, dass DFRN effiziente Pläne berechnet und dass die Verfahren mit Duplikation denen ohne Duplikation überlegen sind [**58**]. Der erste Approximationsalgorithmus wurde 1997 von Munier und Hahnen vorgestellt [**102**]. Die Autoren zeigen, dass der Approximationsfaktor durch $2 - 1/P$ beschränkt ist und dass diese Schranke scharf ist.

3.2. Untere Schranken für die Isoeffizienz von Berechnungen der Pyramide

In diesem Abschnitt betrachten wir die Skalierbarkeit von Anwendungen, deren Aufgabengraph eine Pyramide oder eine umgekehrte Pyramide ist. Zunächst wollen wir diese Graphen genauer definieren.

DEFINITION 3.2.1. Die *Pyramide der Höhe* n ist ein gerichteter azyklischer Graph $\overrightarrow{\triangle}_n = (V_n, \overrightarrow{E}_n)$, wobei $V_n = \{(i, j) \mid 0 \leq i, j < n,\ i + j < n\}$ die Knotenmenge und

$$\overrightarrow{E}_n = \{((i, j), (i + 1, j)) \in V_n \times V_n\} \cup \{((i, j), (i, j + 1)) \in V_n \times V_n\}$$

die Kantenmenge ist. Die Knoten auf *Ebene* k sind alle Knoten $(i, j) \in V$ mit $i + j = k$.

DEFINITION 3.2.2. Es sei $\overrightarrow{G} = (V, \overrightarrow{E})$ ein gerichteter Graph. Der *umgekehrte Graph* $\overleftarrow{G} = (V, \overleftarrow{E})$ von $\overrightarrow{G}$, ist der Graph, der durch Umkehren aller Kantenrichtungen in $\overrightarrow{G}$ entsteht.

Die ersten beiden in Abbildung 3.1.1 dargestellten Graphen sind somit isomorph zu $\overrightarrow{\triangle}_2$ beziehungsweise zu $\overleftarrow{\triangle}_2$. Der Overhead der Berechnung eines Aufgabengraphen entsteht durch die Datenabhängigkeiten und durch Kommunikationskosten. Um eine untere Schranke für den Overhead zu erhalten reicht es aus, die Datenabhängigkeiten zu betrachten. Wir betrachten somit folgende Pläne.

DEFINITION 3.2.3. Es sei $G = (V, E)$ ein gerichteter azyklischer Graph. Die Berechnungszeit eines Knoten $v \in V$ sei gegeben durch $c(v) \in \mathbb{R}^+$. Ein gültiger Plan $S = (p, t)$ für G besteht aus einer Funktion $p : V \to \{1, \ldots, P\}$ und einer Funktion $t : V \to \mathbb{R}_0^+$, wobei p und t die folgenden Bedingungen erfüllen:

(1) Der Graph hat einen Startknoten, das heißt, es gibt einen Knoten $v \in V$ mit $t(v) = 0$.
(2) Alle Datenabhängigkeiten $(v, w) \in E$ werden berücksichtigt, also $t(v) + c(v) \leq t(w)$.
(3) Jeder Prozessor arbeitet immer nur an einer Aufgabe. Es gilt also für alle $v, w \in V$ mit $v \neq w$ und $p(v) = p(w)$, dass $[t(v), t(v) + c(v)] \cap [t(w), t(w) + c(w)] = \emptyset$.

Für den Makespan gilt somit $C_{\max} = \max\{t(v) + c(v) \mid v \in V\}$. Der Overhead ist gegeben durch $T_o = P\, C_{\max} - \sum_{v \in V} c(v)$. Weiterhin erhalten wir durch Umkehren eines Planes für einen Graphen $\overrightarrow{G}$ einen gültigen Plan für seinen umgekehrten Graphen.

LEMMA 3.2.4. *Es sei $\overrightarrow{G} = (V, \overrightarrow{E})$ ein gerichteter azyklischer Graph und $S = (p, t)$ ein gültiger Plan für $\overrightarrow{G}$ mit Makespan C_{max}. Dann ist der Plan $\overline{S} = (p, \overline{t})$ mit $\overline{t}(v) = C_{max} - (t(v) + c(v))$, $v \in V$ ein gültiger Plan für den umgekehrten Graphen von $\overrightarrow{G}$. Die Pläne S und $\overline{S}$ haben denselben Overhead und Makespan.*

BEWEIS. Der Beweis folgt direkt aus Definition 3.2.3. □

Somit ist also insbesondere jede untere Schranke für den Overhead der Berechnung einer Pyramide auch eine untere Schranke für den Overhead der Berechnung der entsprechenden umgekehrten Pyramide.

Um untere Schranken für die Isoeffizienz von Berechnungen von (umgekehrten) Pyramiden zu erhalten, betrachten wir die folgenden drei Fälle.

(1) Jeder Knoten hat die gleiche Berechnungszeit.
(2) Die Berechnungszeiten steigen linear von einer Ebene zur nächsten.
(3) Die Berechnungszeiten sinken linear von einer Ebene zur nächsten.

In allen Fällen ist die Eingabe der Berechnung die Höhe der Pyramide. Die Menge der zulässigen Eingaben ist somit $\mathbb{N}$.

Wir werden zeigen, dass die Isoeffizienzfunktion im ersten Fall P^2 und in den anderen beiden Fällen P^3 dominiert. Die untere Schranke im ersten Fall lässt sich intuitiv wie folgt erklären. In diesem Fall ist der Overhead $T_o(n, P)$, der aus den Datenabhängigkeiten resultiert, quadratisch von P abhängig. Wenn man nun die Effizienz $\frac{1}{1+T_o(n,P)/T(n,1)}$ konstant halten will, so muss die

sequentielle Zeit $T(n,\,1)$ ebenfalls quadratisch in P ansteigen. Die Ergebnisse in den anderen beiden Fällen lassen sich auf ähnliche Weise herleiten. Hier ist der Overhead kubisch in P.

THEOREM 3.2.5. *Wenn jeder Knoten die gleiche Berechnungszeit t benötigt, dann dominiert die Isoeffizienz $I_e(P)$ der Berechnung von (umgekehrten) Pyramiden P^2.*

BEWEIS. Um eine untere Schranke für den Overhead einer Berechnung bei Eingabe $n \in \mathbb{N}$ herzuleiten, reicht es wegen Lemma 3.2.4, die Berechnung einer Pyramide der Höhe n zu betrachten. Der Knoten $(0,\,0)$ muss von mindestens einem Prozessor berechnet werden. Während dieser Zeit können die anderen $P-1$ Prozessoren keine Berechnungen durchführen, die ein sequentieller Algorithmus durchführen würde. Sie tragen somit $(P-1)\,t$ Zeiteinheiten zum Overhead bei. Allgemein gilt, dass in Ebene $l < P$ höchstens $l+1$ Prozessoren beschäftigt sind. Folglich beträgt der in dieser Zeit entstehende Overhead $(P-(l+1))\;t$ Zeiteinheiten. Somit gilt für den Overhead $T_o(n,\,P)$:

$$\begin{aligned} T_o(n,P) &\geq \sum_{l=0}^{\min(n,P)-1} (P-(l+1))\;t \\ &=: \underline{T}_o(n,P) = \begin{cases} \left(nP - \frac{n}{2} - \frac{n^2}{2}\right) t & \text{falls} \quad n < P-1 \\ \frac{P\,(P-1)}{2}\,t & \text{falls} \quad n \geq P-1 \end{cases} . \end{aligned}$$

Die sequentielle Zeit ist gegeben durch $T(n,\,1) = \frac{n\,(n+1)}{2}t$, und

$$\underline{T}(n,P) := \frac{1}{P}\left(T(n,1) + \underline{T}_o(n,P)\right) \tag{3.2.1}$$

ist eine untere Schranke für die parallele Laufzeit. Basierend auf dieser Schranke definieren wir eine obere Schranke für die Effizienz

$$\overline{E}(n,\,P) := \frac{T(n,\,1)}{P\,\underline{T}(n,\,P)} = \begin{cases} \frac{n+1}{2P} & \text{falls} \quad n \leq P \\ 1/\left(1 + \frac{P\,(P-1)}{n\,(n+1)}\right) & \text{falls} \quad n > P \end{cases} .$$

Wie man leicht sieht, ist $\overline{E}(n,\,P)$ für festes P monoton steigend auf $\mathbb{N}$. Daher erfüllt $\overline{E}$ size-up Bedingung.

Es sei $\underline{I}_e(P)$ eine untere Schranke für die Isoeffizienz gemäß Theorem 2.2.3. Falls $\underline{I}_e(P)$ existiert, gibt es wegen Theorem 2.2.5 ein $n \in \mathbb{N}$, so dass

$$\underline{I}_e(P) = T(n,\,1) \geq \frac{e}{1-e}\,\underline{T}_o(n,\,P) \ . \tag{3.2.2}$$

Wir unterscheiden zwei Fälle.

FALL 1. $e = \frac{1}{2}$. In diesem Fall folgt aus der Ungleichung (3.2.2) $n \geq P-1$. Einsetzen in (3.2.2) ergibt

$$\underline{I}_{1/2}(P) \geq \frac{P\,(P-1)}{2}t \succeq P^2 \ .$$

FALL 2. $e \neq \frac{1}{2}$. Da wir das asymptotische Verhalten von $\underline{I}_e(P)$ für $P \to \infty$ betrachten, können wir von $P > \frac{1}{2e-1}$ ausgehen. In diesem Fall gilt $n \geq P-1$. Falls nämlich $n < P-1$ wäre, hätte (3.2.2) die Form $T(n,1) \geq \frac{e}{1-e}\left(nP - \frac{n}{2} - \frac{n^2}{2}\right) t$ — eine Ungleichung, die $n \geq 2eP-1$

impliziert. Dieses würde jedoch wiederum $P \leq \frac{1}{2e-1}$ implizieren. Wegen $n \geq P-1$, folgt aus Gleichung (3.2.2)

$$\underline{I}_e(P) \geq \frac{e}{1-e}\frac{P\,(P-1)}{2}t \succeq P^2 \ .$$

Wegen Theorem 2.2.3 gilt $I_e(P) \geq \underline{I}_e(P)$ und somit $I_e(P) \succeq P^2$. □

THEOREM 3.2.6. *Falls für jedeEbene l die Berechnungszeit durch $(l+1)\,t$ für ein $t > 0$ gebeben ist, dann dominiert die Isoeffizienz $I_e(P)$ der Berechnung von (umgekehrten) Pyramiden P^3.*

BEWEIS. Analog zum Beweis von Theorem 3.2.5 erhalten wir für den Overhead $T_o(n,\,P)$:

$$\begin{aligned} T_o(n,\,P) &\geq \sum_{l=0}^{\min(n,P)-1} (P-(l+1))(l+1)\,t \\ &=: \underline{T}_o(n,P) \\ &= \begin{cases} \left(\frac{nP}{2} - \frac{n}{6} - \frac{n^2}{2} + \frac{n^2P}{2} - \frac{n^3}{3}\right) t & \text{if} \quad n \leq P \\ \left(\frac{P^3}{6} - \frac{P}{6}\right) t & \text{if} \quad n > P \end{cases} . \end{aligned}$$

Die sequentielle Berechnungszeit ist $T(n,\,1) = \left(\frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}\right)\,t$. Es sei $\underline{T}(n,\,P)$ gemäß (3.2.1) definiert. Da $\overline{E}$ die size-up Bedingung erfüllt, gibt es, falls $\underline{I}_e(P)$ existiert, ein $n \in \mathbb{N}$, das die Ungleichung (3.2.2) erfüllt. Wir betrachten zunächst den Fall $e \neq \frac{2}{3}$ und beschränken uns auf $P > \frac{1}{3e-2}$. Damit gilt $n > P$ und Ungleichung (3.2.2) impliziert, dass

$$\underline{I}_e(P) \geq \frac{e}{1-e}\left(\frac{P^3}{6} - \frac{P}{6}\right) t \succeq P^3 \ . \tag{3.2.3}$$

Falls $e = \frac{2}{3}$, folgt aus Ungleichung (3.2.2) $n \geq P$. Es gilt somit für $P > 1$

$$\underline{I}_{2/3}(P) \geq \left(\frac{P^3}{6} - \frac{P}{24}\right) t \succeq P^3 \ .$$

Da $I_e(P) \geq \underline{I}_e(P)$, erhalten wir $I_e \succeq P^3$. □

THEOREM 3.2.7. *Falls für jede Ebene l die Berechnungszeit durch $(n-(l+1))\,t$ für ein $t > 0$ gegeben ist, dann dominiert die Isoeffizienz $I_e(P)$ der Berechnung von (umgekehrten) Pyramiden P^3.*

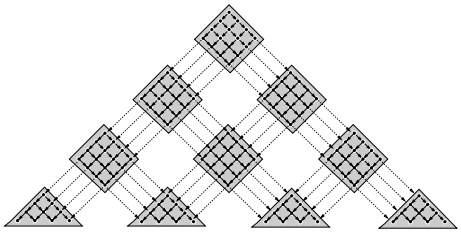


ABBILDUNG 3.3.1. Durch die Gruppierung der Knoten einer Pyramide der Höhe n in Kacheln entsteht eine neue Pyramide der Höhe $n' < n$.

BEWEIS. Für den Overhead erhalten wir

$$\begin{aligned} T_o(n,P) &\geq \sum_{l=0}^{\min(n,P)-1} (P-(l+1))(n-(l+1))\,t \\ &= \begin{cases} \left(\frac{n^2P}{2} - \frac{nP}{2} + \frac{n}{6} - \frac{n^3}{6}\right) t & \text{falls} \quad n \leq P \\ \left(\frac{P(P-1)}{2} n - \frac{P^3}{6} + \frac{P}{6}\right) t & \text{falls} \quad n > P \end{cases} \\ &> \underline{T}_o(n,P) \\ &:= \begin{cases} \left(\frac{n^2P}{2} - \frac{nP}{2} + \frac{n}{6} - \frac{n^3}{6}\right) t & \text{falls} \quad n \leq P \\ \left(\frac{P^3}{3} - \frac{P^2}{2} + \frac{P}{6}\right) t & \text{falls} \quad n > P \end{cases} . \end{aligned}$$

Die sequentielle Berechnungszeit ist in diesem Fall $T(n,1) = \left(\frac{n^3}{6} - \frac{n}{6}\right) t$. Es gilt

$$\underline{I}_e(P) \geq \begin{cases} \frac{e}{1-e}\left(\frac{P^3}{3} - \frac{P^2}{2} + \frac{P}{6}\right) t & \text{falls} \quad e \neq \frac{1}{3} \quad \text{und} \quad P > \frac{1}{3e-1} \\ \left(\frac{P^3}{3} - P^2 + \frac{P}{6}\right) t & \text{falls} \quad e = \frac{1}{3} \quad \text{und} \quad P > 1 \end{cases} .$$

Es ist also $I_e(P) \geq \underline{I}_e(P) \succeq P^3$. □

Wir können nun einige der in der Einleitung zu diesem Kapitel erwähnten Probleme bezüglich ihrer Skalierbarkeit bewerten. So dominiert die Isoeffizienzfunktion für die Berechnung einer optimalen Klammerung eines Matrixprodukts und das Finden einer optimalen Triangulierung eines konvexen Polygons wegen Theorem 3.2.7 P^3, und die Isoeffizienzfunktion der Berechnung des vollständigen Horner-Schemas und eines Taylor-Shifts dominiert wegen Theorem 3.2.5 P^2 [1].

Im nächsten Abschnitt werden wir Abbildungsverfahren angeben, deren Isoeffizienzfunktion kodominant mit P^2 ist. Allerdings liegen bereits sehr einfache Abbildungen in dieser Klasse, so dass es notwendig ist, eine genauere Analyse durchzuführen, um die Verfahren bewerten zu können.

3.3. Abbildungsverfahren

Wir wollen nun das Problem betrachten, eine Pyramide der Höhe n auf P Prozessoren abzubilden. Wir werden davon ausgehen, dass alle Knoten der Pyramide die gleiche Berechnungszeit *t* haben. Eine grundlegende Technik ist die Gruppierung der Knoten zu *Kacheln* wie es Abbildung

[1] Diese Behauptung gilt nur, wenn wir von konstanten Additions- und Multiplikationszeiten ausgehen, wie es beispielsweise bei der Verwendung von Fließkomma-Arithmetik der Fall ist.

3.3.1 zeigt. Durch diesen Vorgang erhalten wir wiederum eine Pyramide der Höhe $n' < n$. Für die Berechnungszeit t' einer Kachel in den Ebenen $0, \ldots, n'-2$ gilt

$$\left\lfloor \frac{n}{n'} \right\rfloor^2 t \leq t' \leq \left\lceil \frac{n}{n'} \right\rceil^2 t$$

und für die Kacheln in der letzten Zeile gilt

$$\frac{1}{2} \left\lfloor \frac{n}{n'} \right\rfloor \left(\left\lfloor \frac{n}{n'} \right\rfloor + 1 \right) \leq t' \leq \frac{1}{2} \left\lceil \frac{n}{n'} \right\rceil \left(\left\lceil \frac{n}{n'} \right\rceil + 1 \right) \ .$$

Wir erhalten also, falls n' ein Teiler von n ist, eine Pyramide, die bis auf die Knoten in der letzten Ebene wiederum aus Knoten mit gleicher Rechenzeit besteht. Die Rechenzeit der Knoten in der letzten Ebene ist etwa halb so groß wie die der anderen Knoten.

Ein naheliegender Ansatz ist, durch Kachelung eine Pyramide der Höhe P zu bilden, diese in P von oben links nach unten rechts verlaufende diagonale Streifen einzuteilen und dann jedem Prozessor einen Streifen zuzuweisen. Gehen wir einmal davon aus, dass P ein Teiler von n ist. Dann beträgt der Overhead dieses Ansatzes

$$T_o(n,\, P) \geq \frac{P\,(P-1)}{2} \left(\frac{n}{P} \right)^2 t = \frac{(P-1)\,n^2}{2P}\, t \ .$$

Für große Werte von P ist der Overhead also ungefähr so groß wie die sequentielle Laufzeit $n\,(n+1)\,t/2$. Asymptotisch beträgt somit die maximale mit diesem Ansatz erreichbare Effizienz $\frac{1}{2}$.

Die Kachelungstechnik alleine ermöglicht also keine effiziente Abbildung. Allerdings ist sie sehr nützlich, um die Granularität der Berechnung zu erhöhen. Da der Ergebnisgraph ebenfalls eine Pyramide ist, kann sie als Vorstufe für andere Abbildungsverfahren benutzt werden.

Wir werden im Folgenden untere und obere Schranken für die Dauer einer Pyramidenberechnung betrachten, indem wir jeweils den *kritischen Pfad* der Rechnung untersuchen. Der kritische Pfad ist eine Kette unnmittelbar aufeinanderfolgender voneinander abhängender Rechen- und Kommunikationsereignisse. Das erste Ereignis des Pfades beginnt zum Zeitpunkt 0, und die Dauer des letzten Ereignisses bestimmt die Dauer der Gesamtrechnung. Die Summe der Zeiten, die durch die Operationen auf dem kritischen Pfad benötigt werden, ist identisch mit der Gesamtdauer der Berechnung.

Wir leiten untere Schranken für die Berechnungszeit her, indem wir eine untere Schranken für die Summe der Zeiten, die durch Rechenoperationen (T_{calc}), Aufsetzzeiten (T_{ovhd}) und Latenzzeiten (T_{lat}) entstehen, herleiten. Obere Schranken für die Berechnungszeiten werden mit Hilfe des folgenden Lemmas hergeleitet.

LEMMA 3.3.1. *Es sei $T(L,\, o,\, P,\, t)$ die Auswertungszeit eines Aufgabengraphen mit Aufgabenberechnungszeit t, Latenzzeit L und Aufsetzzeit o. Dann gilt*

$$T(L,\, o,\, P,\, t) \leq T(L,\, 0,\, P,\, 0) + T(0,\, o,\, P,\, 0) + T(0,\, 0,\, P,\, t) \ .$$

Für die Berechnung einer Pyramide der Höhe n gilt zusätzlich

$$T(L,\, o,\, P,\, 0) \leq (n-1)\,L \ .$$

BEWEIS. Angenommen, die Behauptung würde nicht gelten. Nehmen wir außerdem noch an, der kritische Pfad der Berechnung mit den Parametern $(L,\, o,\, P,\, t)$ enthalte mehr Rechenoperationen, als der von $(0,\, 0,\, P,\, t)$. Da die Rechenoperationen im kritischen Pfad von $(L,\, o,\, P,\, t)$ voneinander abhängen, existiert diese Folge ebenfalls in der Berechnung mit den Parametern $(0,\, 0,\, P,\, t)$. Das ist ein Widerspruch zu der Annahme, dass der Pfad von $(0,\, 0,\, P,\, t)$ weniger

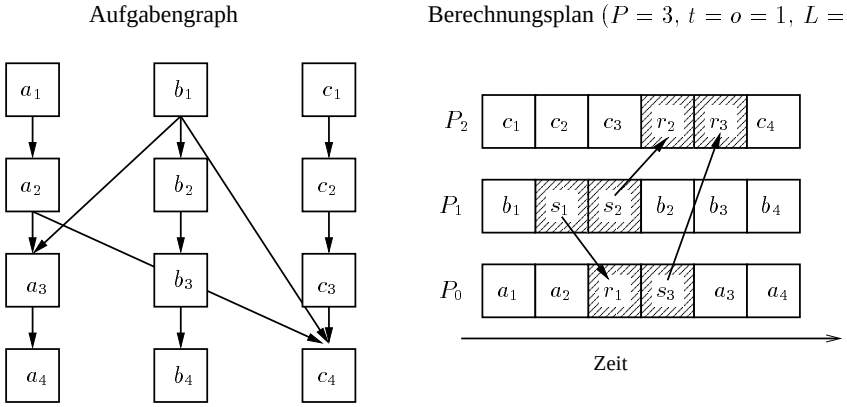


ABBILDUNG 3.3.2. Ein Aufgabengraph und ein Ablaufplan für drei Prozessoren. Beispiele für kritische Pfade in diesem Ablauf sind $(b_1,\ s_1,\ r_1,\ s_3,\ r_3,\ c_4)$ und $(c_1,\ c_2,\ c_3,\ r_2,\ r_3,\ c_4)$. Dagegen besteht die Ereigniskette $(c_1,\ c_2,\ c_3,\ b_2,\ b_3,\ b_4)$ zwar aus aufeinanderfolgenden Ereignissen, sie ist aber kein kritischer Pfad, da die Ereignisse c_3 und b_2 nicht voneinander abhängen.

Rechenoperationen enthält, als der von $(L,\ o,\ P,\ t)$. Auf analoge Weise ist einzusehen, dass der Pfad von $(L,\ o,\ P,\ t)$ ebenfalls nicht mehr Aufsetz- oder Latenzzeiten enthalten kann als der von $(L,\ o,\ P,\ 0)$ bzw. von $(L,\ 0,\ P,\ 0)$.

Die zweite Aussage folgt aus der Tatsache, dass jeder kritische Pfad in einer Pyramide aus höchstens $n - 1$ Kommunikationen besteht. □

3.3.1. Pipelining. Wir betrachten Pyramiden, deren Höhe n ein Vielfaches der Prozessoranzahl P ist. Es sei $s := n/P$. Die Pipelining Methode teilt die Pyramide in s von oben links nach unten rechts verlaufende diagonale Zonen ein. Jede Zone besteht aus P Streifen, wobei jeder Prozessor einen Streifen in jeder Zone bearbeitet. Es entsteht somit die in Abbildung 3.1.3(b) gezeigte Aufteilung. Jeder Prozessor verarbeitet s Streifen.

THEOREM 3.3.2. *Es sei $s \in \mathbb{N}$ und es sei $n = s\,P$. Für den Makespan $T(n,\ P)$ einer mit der Pipelining Methode geplanten Berechnung einer Pyramide der Höhe n auf P Prozessoren gilt*

$$T(n,\ P) \geq \frac{s\,(s+1)}{2}\,P\,(t + 2o) - (s + s\,P)\,o + (P-1)\,L\ .$$

BEWEIS. Betrachten wir zunächst Prozessor 0. Der Streifen von Prozessor 0 in Zone $i \in \{0, \ldots, s-1\}$ besteht aus $(s-i)\,P$ Knoten. Für die Berechnung dieser Knoten wird somit die Zeit

$$T_{\text{calc}}(n,\ P) = \sum_{i=0}^{s-1}(s-i)\,P = \frac{s\,(s+1)}{2}\,P\,t$$

benötigt. Mit Ausnahme des Streifens in der ersten Zone muss Prozessor 0 die Ergebnisse von Prozessor $P-1$ empfangen. Außerdem muss er mit Ausnahme des letzten Knotens eines jeden Streifens die Ergebnisse an Prozessor 1 weiterschicken. Insgesamt erhalten wir somit für den Kommunikationsaufwand

$$T_{\text{ovhd}}(n,\, P) = s\,(s+1)\,P\,o - s\,P\,o - s\,o \;\;.$$

Schließlich berücksichtigen wir noch die Wartezeiten, die bei der Bearbeitung des letzten Streifens auftreten. Falls Prozessor 0 den letzten Streifen zum Zeitpunkt τ beginnt, kann Prozessor 1 seinen letzten Streifen frühestens zum Zeitpunkt $\tau + 2o + L$ beginnen. Wir erhalten also zusätzlich mindestens die Wartezeit von

$$\underline{T}_{\text{lat}}(n,\, P) = (P-1)\,L \;\;.$$

Folglich ist $T_{\text{calc}}(n,\, P) + T_{\text{ovhd}}(n,\, P) + \underline{T}_{\text{lat}}(n,\, P)$ eine untere Schranke für die Berechnungszeit der Pyramide. □

Aus dieser Laufzeitschranke ergibt sich für den Overhead, der durch die Pipelining Abbildung verursacht wird,

$$T_{\text{o}}(n,\, P) \geq \frac{1}{2}\left(s\,P^2 - s\,P\right)\,t + \left(s^2\,P^2 - s\,P\right)\,o + \left(P^2 - P\right)\,L \;\;.$$

Zu beachten sind hierbei vor allem zwei Eigenschaften. Erstens entsteht durch Wartezeiten am Anfang jedes Streifens selbst ohne Kommunikationskosten bei dieser Methode ein Overhead, der $s\,P^2$ dominiert. Zweitens ist der Overhead, der durch die Kommunikation verursacht wird, quadratisch in der Pyramidenhöhe. Trotzdem ist die Isoeffizienz des Pipelining Verfahrens kodominant mit P^2.

THEOREM 3.3.3. *Es sei $e \in (0,\, t/(t+2o))$. Dann ist die Isoeffizienz $I_e(P)$ des Pipelining Verfahrens für alle $P > 1$ definiert und kodominant mit P^2.*

BEWEIS. Es gilt $T_{\text{lat}}(n,\, P) \leq (n-1)\,L$ und somit ist

$$\overline{T}(n,\, P) := \frac{1}{2}\left(\frac{n^2}{s} + n\right)t + \left(\frac{n^2}{s} + n\right)o + n\,L$$

eine obere Schranke für die parallele Laufzeit einer mit Pipelining geplanten Berechnung. Hierdurch erhalten wir eine untere Schranke für die Effizienz durch

$$\underline{E}(n,\, P) := \frac{\overline{T}(n,\, P)}{P\,T(n,\, 1)} = \frac{(n+1)\,t}{(n+P)\,t + 2\,(n+P)\,o + 2\,LP} \;\;.$$

Es sei $\overline{I}_e(P)$ die Isoeffizienzfunktion bezüglich $\underline{E}$. Die Funktion $\underline{E}$ ist monoton steigend in n und somit gibt es für jede Effizienz $e < \lim_{n\to\infty} \underline{E}(n,\, P) = t/(t+2o)$ ein $n \in \mathbb{N}$ mit $\underline{E}(n,\, P) \geq e$. Da $\underline{E}$ die size-up Bedingung erfüllt, gilt $\overline{T}(n,\, P) = \overline{I}_e(P) \geq I_e(P)$.

Die Ungleichung $E(n,\, P) \geq e$ ist äquivalent zu

$$n \geq \left\lceil \frac{e\,P\,(t+2o) + 2\,e\,P\,L - t}{e\,(t+2\,o) - t} \right\rceil =: n_0 \;\;.$$

Folglich gilt $\overline{I}_e(P) = T(n_0,\, 1)$. Da $n_0 \sim P$ und $T(n,\, 1) \sim n^2$, folgt $I_e(P) \leq \overline{I}_e(P) \preceq P^2$. Wegen Theorem 3.2.5 gilt $I_e(P) \succeq P^2$ und somit gilt insgesamt $I_e(P) \sim P^2$. □

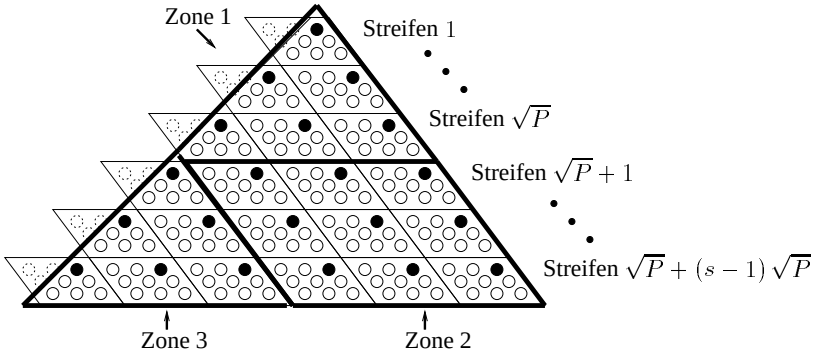


ABBILDUNG 3.3.3. ZS-Mapping einer Pyramide der Höhe 18 auf 9 Prozessoren. Die Knoten, die von Prozessor 0 verarbeitet werden, sind schwarz gekennzeichnet. Die Zonen beziehen sich auf die dick umrandeten Gebiete.

3.3.2. Zyklisches Schachbrettmapping. Das zyklische Schachbrettmapping (ZS-Mapping) ist eng mit der systolischen Programmierung verwandt. Es teilt die Pyramide in Zellen mit jeweils P Knoten auf. Jeder Prozessor ist jeweils für einen Knoten in jeder Zelle verantwortlich. Abbildung 3.3.3 zeigt die hierdurch entstehende Einteilung in waagerechte Streifen. Die Prozessoren bearbeiten die Zellen streifenweise jeweils von rechts nach links.

THEOREM 3.3.4. *Es sei $s,\, k \in \mathbb{N}$ und es sei $n = s\,P = s\,k^2$. Für den Makespan $T(n,\,P)$ einer mit ZS-Mapping geplanten Berechnung einer Pyramide der Höhe n auf P Prozessoren gilt*

$$\begin{aligned} T(n,\,P) \;\geq\; & \frac{1}{2}\left(\left(s^2+1\right)\,P+(s+1)\,\sqrt{P}-2\right)\,(t+4o)\,- \\ & \left(P+(2s-1)\,\sqrt{P}+2\right)\,o+\left(P+\sqrt{P}-1\right)\,L\;. \end{aligned}$$

BEWEIS. Zunächst betrachten wir die Zeit, die benötigt wird, um alle Knoten unter Vernachlässigung der Kommunikationskosten zu berechnen. Bevor Prozessor 0 den zweiten Streifen beginnen kann, muss zunächst die Berechnung der Zelle in Streifen 1 abgeschlossen sein. Hierdurch entsteht für Prozessor 0 eine Wartezeit von $(\sqrt{P}-1)\,t$ Zeiteinheiten. Da Prozessor 0 im zweiten Streifen zwei Knoten zu berechnen hat, verkürzt sich diese Wartezeit beim dritten Streifen auf $(\sqrt{P}-2)\,t$. Insgesamt werden also für die Berechnung der ersten $\sqrt{P}$ Streifen (Zone 1 in Abbildung 3.3.3) $P\,t$ Zeiteinheiten benötigt. Bei der Bearbeitung der Knoten in verbleibenden $(s-1)\,\sqrt{P}$ Streifen (Zonen 2 und 3) kommt es zu keinen Wartezeiten mehr. Wenn wir zusätzlich noch die Bearbeitungszeit der letzten Zelle in Streifen $s\,\sqrt{P}$ berücksichtigen, erhalten wir

$$T_{\text{calc}}(n,\,P) =$$

$$= \left(\underbrace{P}_{\text{Zone 1}} + \underbrace{(s-1)\,\sqrt{P}\cdot\sqrt{P}}_{\text{Zone 2}} + \underbrace{\frac{(s-1)\,\sqrt{P}\,\left((s-1)\,\sqrt{P}+1\right)}{2}}_{\text{Zone 3}} + \underbrace{\sqrt{P}-1}_{\text{letzte Zelle}} \right)\,t$$

$$= \frac{1}{2}\left(\left(s^2+1\right)\,P+(s+1)\,\sqrt{P}-2\right)\,t\;.$$

Alle Knoten im Innern der Pyramide erfordern 2 Empfangs- und 2 Sendeoperationen. Die Knoten an den Rändern benötigen dagegen weniger Empfangsoperationen. Im einzelnen erhalten wir

$$T_{\text{ovhd}}(n,\,P) \;=\; 2\left(\left(s^2+1\right)\,P+(s+1)\,\sqrt{P}-2\right)\,o\,- \\ \left(\underbrace{P}_{\text{Zone }1}+\underbrace{1}_{\text{Zelle }1}+\underbrace{2\,(s-1)\sqrt{P}}_{\text{Zonen }2,\,3}+\underbrace{\sqrt{P}-1+2}_{\text{letzte Zelle}}\right)\,o\;.$$

Durch die Latenzzeit entstehen Wartezeiten mindestens in Zone 1 und während der Berechnung der letzten Zelle. Also, $\underline{T}_{\text{lat}}(n,\,P) \;=\; (P+\sqrt{P}-1)\,L$. Schließlich ergibt sich die untere Schranke für $T(n,\,P)$ als Summe der Zeiten $T_{\text{calc}}(n,\,P)$, $T_{\text{ovhd}}(n,\,P)$ und $\underline{T}_{\text{lat}}(n,\,P)$. □

Für den gesamten Overhead erhalten wir

$$T_{\text{o}}(n,\,P) \;\geq\; \frac{1}{2}\left((s+1)\,P\,\sqrt{P}+(P-s-2)\,P\right)\,t\,+ \\ \left(2\,s^2\,P^2+3\,P\,\sqrt{P}+P^2-6P\right)\,o+\left(P^2+P\sqrt{P}-P\right)\,L\;.$$

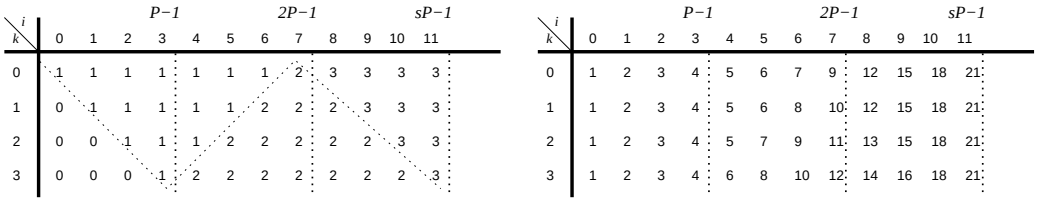
Der Overhead, der bereits ohne Berücksichtigung der Kommunikationskosten entsteht, ist kodominant mit $s\,P\,\sqrt{P}+P^2$ und ist somit kleiner als der entsprechende Overhead der Pipelining Methode. Allerdings wächst er ebenso wie bei der Pipelining Methode mit wachsender Pyramidenhöhe. Andererseits sind die Kommunikationskosten von ZS Mapping höher als bei Pipelining. Insgesamt führt also ZS-Mapping nur bei grobgranularen Rechnungen zu kürzeren Berechnungszeiten als Pipelining.

3.3.3. Pie-Mapping. Die Skalierbarkeit der beiden in den vorangegangenen Abschnitten beschriebenen Methoden wird durch zwei Faktoren beeinträchtigt. Zum einen erfordert die Bearbeitung jedes Knotens das Senden oder das Empfangen von Nachrichten. Hierdurch entsteht ein Kommunikationsoverhead, der quadratisch in der Pyramidenhöhe ist. Zum anderen ist der Overhead, der bereits ohne Kommunikationskosten entsteht, bei beiden Verfahren von der Pyramidenhöhe abhängig.

Das Pie-Mapping Verfahren teilt die Pyramide in zusammenhängende Zonen ein, die jeweils von einem Prozessor bearbeitet werden (vgl. Abbildung 3.1.3(c)). Dadurch können die meisten Knoten berechnet werden, ohne Daten empfangen oder versenden zu müssen. Desweiteren ist der Overhead ohne Kommunikationskosten nur von der Prozessorzahl abhängig.

Wir nehmen wieder an, dass die Pyramidenhöhe durch die Prozessorzahl teilbar ist. Es ist also $n = s\,P$ für ein $s \in \mathbb{N}$. Wir definieren durch eine *Lastmatrix*, wieviele Knoten ein Prozessor jeweils in einer bestimmten Ebene bearbeitet. Mit Hilfe dieser Lastmatrix ist das Pie-Mapping eindeutig definiert. Um einen gültigen Plan für die Pyramide zu erhalten, ist es notwendig, die Reihenfolge der Knotenauswertung innerhalb der Zonen zu definieren. Hierfür werden wir zwei Reihenfolgen untersuchen. Die erste sieht eine ebenenweise Abarbeitung und die zweite eine in Blöcken organisierte Abarbeitung vor. In beiden Fällen werden wir zeigen, dass bei Vernachlässigung der Kommunikationskosten ein Prozessor, nachdem er mit der Bearbeitung seiner Knoten begonnen hat, nicht mehr auf andere Prozessoren warten muss, und dass, wenn s ungerade ist, alle Prozessoren zur gleichen Zeit fertig werden. Es wird sich allerdings herausstellen, dass bei Berücksichtigung der Kommunikationskosten die Blockreihenfolge effizienter ist.

Wir benutzen eine Lastmatrix $L(P,\,s)$ für die Beschreibung der Zonenaufteilung der Abbildung einer Pyramide der Höhe $s\,P$ auf P Prozessoren. Der Eintrag $l_{k,i}$ definiert die Anzahl der



Lastmatrix Akkumulationsmatrix

ABBILDUNG 3.3.4. Beispiele einer Last- und einer Akkumulationsmatrix für den Fall $P = 4,\ s = 3$. Die Akkumulationsmatrix bezieht sich auf die ebenenweise Abarbeitung der Knoten.

Knoten, die Prozessor $P - k - 1$ in der Ebene i bearbeitet. In Abbildung 3.3.4 ist die Lastmatrix $L(4,\ 3)$ dargestellt. Der Verlauf der diagonalen Linien legt eine Definition der Matrix mittels der folgenden Matrizen nahe.

DEFINITION 3.3.5. Es sei k eine natürliche Zahl. Dann sind die Matrizen $O_{P,k}, U_{P,k} \in \mathbb{N}^{P\times P}$ wie folgt definiert. Die Einträge der Matrix $O_{P,k}$ sind gleich k auf und *oberhalb* der Diagonalen und $k - 1$ sonst. Die Einträge der Matrix $U_{P,k}$ sind gleich k *unterhalb* und auf der Gegendiagonalen und $k - 1$ sonst.

Bezüglich der Matrix $L(4,\ 3)$ gilt also

$$L(4,\ 3) = (O_{4,1}\, U_{4,2}\, O_{4,3}) \quad .$$

Allgemein definieren wir die Matrix $L(P,\ s)$ wie folgt.

DEFINITION 3.3.6. Es sei $P,\ s \in \mathbb{N}$. Die *Lastmatrix* $L(P,\ s) = (l_{k,i})_{\substack{k=0,\ldots,P-1\\ i=0,\ldots,sP-1}}$ ist definiert durch

$$L(P,\ s) = \begin{cases} (O_{P,1}\, U_{P,2}\, O_{P,3}\, U_{P,4} \cdots O_{P,s}) & \text{falls} \quad s \text{ ungerade,} \\ (O_{P,1}\, U_{P,2}\, O_{P,3}\, U_{P,4} \cdots U_{P,s}) & \text{falls} \quad s \text{ gerade.} \end{cases}$$

Aus der Konstruktion der Lastmatrix folgt unmittelbar die folgende Eigenschaft.

LEMMA 3.3.7. *Es sei $L(P, s) = (l_{k,i})$ eine Lastmatrix. Dann gilt für $0 \le i < sP - 2$ und $0 \le k < P - 1$ die Beziehung*

$$l_{k+1,i} \le l_{k,i+2} \quad .$$

BEWEIS. Per Konstruktion gilt die Beziehung $l_{k+1,i} \le l_{k,i+1}$. Damit folgt die Behauptung aus $l_{k,i+1} \le l_{k,i+2}$. □

Im Folgenden bezeichnen wir die Ebenen $(i - 1)\, P, \ldots, i\, P - 1$ als *Phase* i. Die Ebenen innerhalb einer Phase bezeichnen wir als *Stufen.* Somit ist die Ebene $(i - 1)\, P + j$ die j-te Stufe in Phase i.

DEFINITION 3.3.8. Es sei $P,\ s > 0$ und es sei $L(P,\ s) = (l_{ki})$ die Lastmatrix. Das *Pie-Mapping* für P und s bildet den Knoten $(r,\ c)$ auf den Prozessor $P - k - 1$ ab, wobei k die kleinste ganze Zahl ist mit $r < \sum_{j=0}^{k} l_{j,r+c}$.

Um auf der Basis des Pie-Mappings zu einem vollständigen Plan zu gelangen, ist noch die Abarbeitungsreihenfolge festzulegen. Wir betrachten zunächst eine einfache ebenenweise Berechnung.

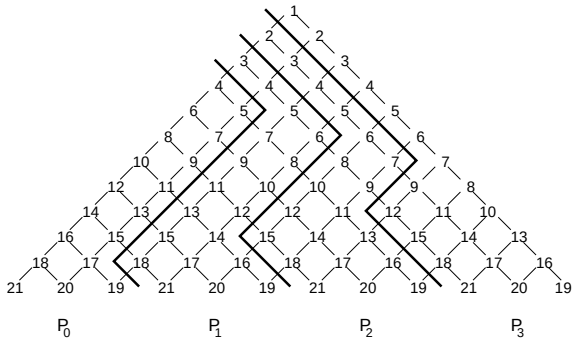


ABBILDUNG 3.3.5. Pie-Mapping mit Ebenenplanung für 4 Prozessoren und $s = 3$. Die Abbildung zeigt die Abhängigkeiten zwischen den Knoten sowie das Mapping der Knoten auf die Prozessoren. Ebenfalls angegeben sind die Berechnungszeiten der einzelnen Knoten für den Fall $L = o = 0$ und $t = 1$. Die Anzahl der Knoten, die Prozessor k in Ebene i abarbeitet, sind durch den Eintrag $l_{P-k-1,i}$ in der Lastmatrix $L(4,\ 3)$ gegeben. Die Zeit, zu der Prozessor k die Ebene i beendet, ist durch den Eintrag $a_{P-k-1,i}$ der Akkumulationsmatrix $A(4,\ 3)$ gegeben (Lemma 3.3.11).

3.3.4. Ebenenplanung. Bei der *Ebenenplanung* werden die Knoten einer Zone Ebene für Ebene von rechts nach links berechnet. Abbildung 3.3.5 zeigt die Berechnungszeiten, die beim Pie-Mapping mit Ebenenplanung für 4 Prozessoren und $s = 3$ entstehen. Um die Blockierungsfreiheit zu untersuchen, definieren wir die Akkumulationsmatrix.

DEFINITION 3.3.9. Es sei $P,\ s > 0$ und es sei $L(P,\ s)$ die Lastmatrix. Die *Akkumulationsmatrix* $A(P,\ s)$ ist definiert durch

$$A(P,\ s) = (a_{k,i})\ \begin{matrix} k = 0, \ldots, P-1 \\ i = 0, \ldots, sP-1 \end{matrix}\ \text{, wobei } a_{k,i} = \sum_{j=0}^{i} \max(1,\ l_{k,j})\ .$$

Abbildung 3.3.4 zeigt die Akkumulationsmatrix für den Fall $P = 4,\ s = 3$. Die Akkumulationsmatrix hat die folgende Eigenschaft.

LEMMA 3.3.10. *Es sei $L(P,s) = (l_{k,i})$ eine Lastmatrix und $A(P,s) = (a_{k,i})$ die zugehörige Akkumulationsmatrix. Dann gelten für $0 \leq k < P-1$ die Beziehungen*

(1) $a_{k-1,i} < a_{k,i} + 1$ *für* $0 \leq i \leq sP-1$ *und* $0 < k \leq P-1$,
(2) $a_{k+1,i-1} + 1 < a_{k,i+1}$ *für* $0 < i < sP-1$ *und* $0 \leq k < P-1$.

BEWEIS. (1) Wir können ohne Einschränkung davon ausgehen, dass s ungerade ist. Falls s gerade ist, benutzen wir die Aussage des Lemmas für den Fall $s+1$.

Es sei $0 < k \leq P-1$. Wir beweisen die Aussage durch Induktion. Sie ist richtig für $0 \leq i \leq P-1$. Es sei nun für ein ungerades l mit $1 \leq l < s$ bewiesen, dass die Behauptung für $0 \leq i \leq lP-1$ richtig ist. Nun betrachten wir die Matrix $(U_{P,l+1}\ O_{P,l+2})$. Die Zeilen $k-1$ und k dieser Matrix haben die Form

$$\begin{array}{cccccccccc} \cdots & l & l & l+1 & \cdots & l+1 & l+2 & l+2 & \cdots \\ \cdots & l & l+1 & l+1 & \cdots & l+1 & l+1 & l+2 & \cdots \end{array}\ .$$

Da die Einträge $a_{k-1,i}$ und $a_{k,i}$ durch Summierung der Einträge in diesen Zeilen von links nach rechts entstehen, gilt für $lP - 1 \leq i \leq (l+2)\,P - 1$ die Beziehung $a_{k-1,i} \leq a_{k,i}$.

(2) Wegen der Konstruktion der Akkumulationsmatrix gilt die zweite Behauptung für $i = 1$. Es gelte nun $a_{k+1,i-1} + 1 < a_{k,i+1}$ für ein $i \geq 1$. Dann folgt wegen Lemma 3.3.7,

$$a_{k+1,i} + 1 = a_{k+1,i-1} + l_{k+1,i} + 1 < a_{k,i+1} + l_{k+1,i} \leq a_{k,i+1} + l_{k,i+2} = a_{k,i+2} \ .$$

□

LEMMA 3.3.11. *Eine Pyramide der Höhe $s\,P$ werde durch Ebenenplanung auf P Prozessoren abgebildet. Es sei $A(P,\,s) = (a_{k,i})$ die Akkumulationsmatrix. Wenn alle Knoten die Berechnungszeit 1 haben und die Kommunikationskosten vernachlässigt werden, dann beendet Prozessor k die Ebene i zum Zeitpunkt $a_{P-k-1,i}$. Insbesondere beenden alle Prozessoren die letzte Ebene zum Zeitpunkt $((s^2+1)P + s - 1)/2$ wenn s ungerade ist. Wenn s gerade ist, dann endet die Berechnung zum Zeitpunkt $((s^2+2)P + s - 2)/2$.*

BEWEIS. Der Beweis erfolgt durch Induktion bezüglich der Pyramidenebene. Da in den ersten P Ebenen pro Ebene ein weiterer Prozessor benutzt wird, verläuft die Berechnung in der ersten Phase blockierungsfrei und die Aussage des Lemmas gilt. Es sei die Behauptung des Lemmas für ein $i > 0$ und $0 \leq k \leq P - 1$ gezeigt. Es sei k mit $0 \leq k \leq P$ beliebig aber fest. Weiterhin sei $(a,\,b)$ ein Knoten in Ebene $i+1$, der von Prozessor $P - k - 1$ berechnet wird. Der Knoten benötigt Daten von den Knoten $(a-1,\,b)$ und $(a,\,b-1)$ (falls diese Knoten existieren). Wir betrachten zunächst den Knoten $(a-1,\,b)$. Es kann nur dann zu Verzögerungen kommen, wenn dieser Knoten von Prozessor $P - k - 2$ berechnet wird. In diesem Fall ist der Knoten $(a,\,b)$ der letzte Knoten, den Prozessor $P - k - 1$ in Ebene $i+1$ berechnen muss. Nach Induktionsvoraussetzung ist der Knoten $(a-1,\,b)$ zum Zeitpunkt $a_{k+1,i-1}+1$ berechnet. Wegen Lemma 3.3.10(2) gilt $a_{k+1,i-1} + 1 < a_{k,i+1}$, und folglich kann die Berechnung von $(a,\,b)$ zum geforderten Zeitpunkt $a_{k,i+1}$ abgeschlossen werden.

Falls der Knoten $(a,\,b-1)$ nicht von Prozessor $P - k - 1$, sondern von Prozessor $P - k$ bearbeitet wird, dann ist der Knoten $(a,\,b)$ der erste Knoten, der von Prozessor $P-k-1$ in Ebene $i+1$ zu berechnen ist. Der Prozessor $P - k$ berechnet den Knoten $(a,\,b-1)$ als letzten Knoten in Ebene i nach Voraussetzung zum Zeitpunkt $a_{k-1,i}$. Weiterhin beendet Prozessor $P-k-1$ die Ebene i zum Zeitpunkt $a_{k,i}$. Da nach Lemma 3.3.10(1) $a_{k-1,i} < a_{k,i} + 1$ gilt, kann der Prozessor $P - k - 1$ die Ebene $i+1$ ohne Verzögerung beginnen.

Da die Zeitpunkte, zu denen die Berechnung der Ebenen in den einzelnen Zonen abgeschlossen ist, durch die Akkumulationsmatrix gegeben sind, ist schließlich die Gesamtberechnungszeit gleich dem Eintrag $a_{P-1,sP-1}$. □

Wir erhalten somit für den Overhead bei Vernachlässigung der Kommunikationskosten

$$T_{\mathrm{o}}(n,\,P) = \begin{cases} \frac{P^2-P}{2}\,t & \text{falls} \quad s \text{ ungerade} \\ (P^2 - P)\,t & \text{falls} \quad s \text{ gerade.} \end{cases}$$

In beiden Fällen ist der Overhead nicht von der Pyramidenhöhe abhängig. Abbildung 3.3.6 zeigt ein Ausführungsprofil einer Berechnung einer Pyramide der Höhe 8000 auf 16 Prozessoren. Mit Hilfe der Kachelungstechnik wurde zunächst eine Pyramide der Höhe 48 ($s = 3$) erzeugt. In Phase 1 beginnen die Prozessoren nacheinander mit der Rechnung. In den Phasen $i = 2,\,3$ berechnen die Prozessoren bis zu i Knoten pro Ebene. Deutlich ist zu erkennen, dass die Prozessoren in der Tat die ungeraden Phasen (nahezu) gleichzeitig beenden. In diesem Beispiel hat der

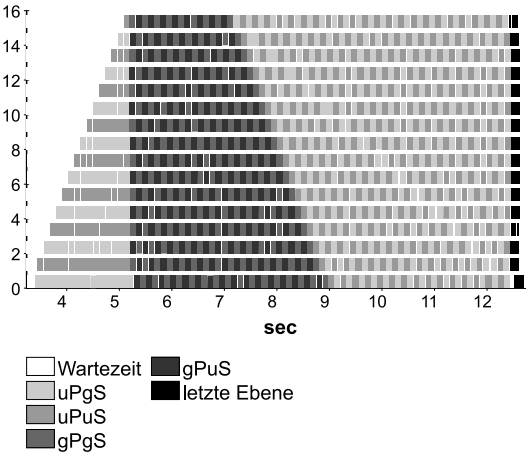


ABBILDUNG 3.3.6. Pie-Mapping mit Ebenenplanung einer Pyramide der Höhe 48. In dem Ausführungsprofil ist auf der x-Achse die Zeit aufgetragen. Die waagerechten Streifen zeigen die Aktivitäten der Prozessoren. Die weißen Flächen repräsentieren Wartezeiten. Die Abkürzungen in der Legende stehen für "gerade Phase, gerade Spalte", "gerade Phase, ungerade Spalte" und so weiter, wobei die Spalte eines Knotens $(i,\ j)$ gerade druch j gegeben ist. Die Messungen wurde auf dem Parsytec CC durchgeführt.

Kommunikationsoverhead kaum Einfluss auf die parallele Rechenzeit. Dieses ändert sich, wenn die Höhe der Ausgangspyramide kleiner ist und somit die Berechnung feingranularer wird.

Im Folgenden wollen wir den Overhead, der durch die Kommunikationskosten entsteht, genauer untersuchen. Hierfür betrachten wir den Fall, in dem die Latenzzeit und die Berechnungszeit 0 sind. Der einzige Aufwand entsteht also durch den Kommunikationsoverhead. Abbildung 3.3.7 zeigt die Berechnungszeiten für 6 Prozessoren und $s = 5$. Außer bei den beiden äußeren Prozessoren steigen die Berechnungszeiten am Ende der dritten Phase von rechts nach links um jeweils einen Zeitschritt an. In den folgenden Phasen kommt es bei den Prozessoren $1, \ldots P-2$ zu keinen weiteren Wartezeiten mehr. Da der Kommunikationsaufwand dieser Prozessoren gleich ist, bleibt es bei diesem Anstieg der Berechnungszeiten am Ende jeder Phase. Es kommen also ab einschließlich Phase 4 pro Phase $2P$ Kommunikationsvorgänge dazu. Insgesamt erhalten wir für die Gesamtberechnungszeit $T_{\text{ovhd}}(n,\ P) = (2s+1)\,P - 1$.

Da für die Analyse insbesondere die Berechnungszeiten an den Zonengrenzen von Interesse sind, führen wir die Bezeichnung $l_{i,j}^{(k)}$ für die Berechnungszeit des Knotens an der linken Grenze in der Stufe $j = 0, \ldots, P-1$ innerhalb der Phase $i = 1, \ldots, s$ von Prozessor $k = 0, \ldots, P-1$ ein. Entsprechend bezeichnen wir mit $r_{i,j}^{(k)}$ die Berechnungszeiten an der rechten Grenze von Prozessor k. Abbildung 3.3.8 verdeutlicht dieses Indexierungsschema. Die Zeiten in der Phase 1 werden mit dem Schema für ungerades i bezeichnet, allerdings sind hier die Zeiten $l_{1,j}^{(k)}$ und $r_{1,j}^{(k)}$ für $0 \leq j \leq P-k-2$ undefiniert.

LEMMA 3.3.12. *Eine Pyramide der Höhe $3\,P$ werde durch Ebenenplanung auf $P \geq 3$ Prozessoren abgebildet. Wenn die Berechnungszeit sowie die Latenzzeit vernachlässigt werden und*

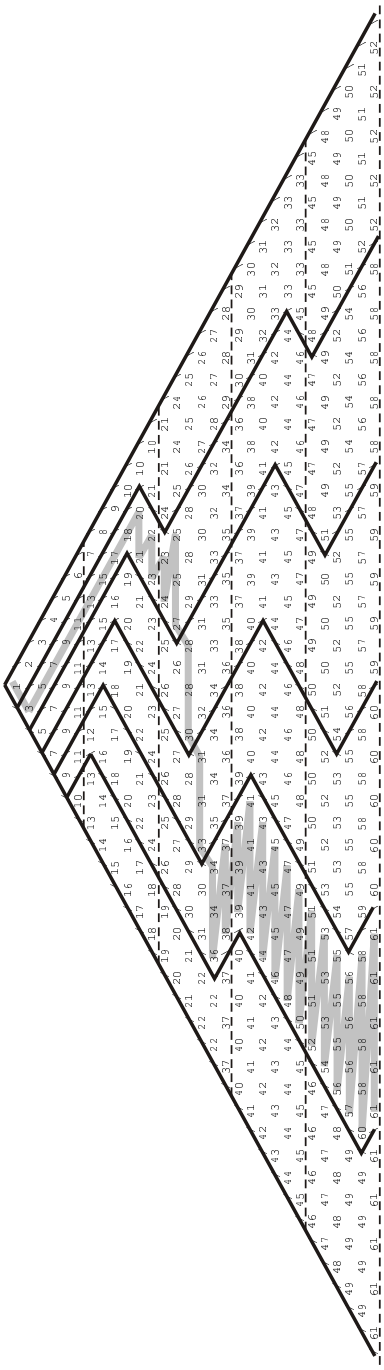


ABBILDUNG 3.3.7. Berechnungszeiten bei Ebenenplanung für den Fall $P = 6$, $s = 5$, $L = t = 0$ und $o = 1$. Der graue Streifen zeigt den Verlauf des kritischen Pfades. Ab einschließlich Phase 4 entstehen durch den Kommunikationsoverhead nur noch Wartezeiten bei den beiden äußeren Prozessoren, die jedoch nicht auf dem kritischen Pfad liegen.

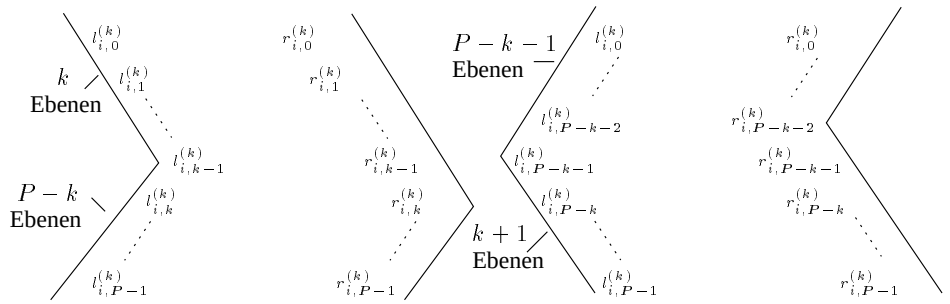


ABBILDUNG 3.3.8. Die Berechnungszeiten an den Kanten werden durch die Werte $l^{(k)}_{i,j}$ und $r^{(k)}_{i,j}$ für $i \in \{1,\ldots,s\},\ j,k \in \{0,\ldots,P-1\}$ dargestellt.

der Kommunikationsoverhead $o = 1$ ist, dann gilt

$$l^{(k)}_{3,P-1} = r^{(k)}_{3,P-1} = \begin{cases} 7\,P-5 & \textit{falls} \quad k=0 \\ 7\,P-k-4 & \textit{falls} \quad 0<k<P-1 \\ 5\,P-2 & \textit{falls} \quad k=P-1\ . \end{cases}$$

BEWEIS. *Phase 1*: In dieser Phase gilt stets $l^{(k)}_{1,j} = r^{(k)}_{1,j}$. Prozessor $P-1$ versendet während dieser Phase P Botschaften. Es gilt daher $l^{(P-1)}_{1,P-1} = P$. Jeder Prozessor $k \in \{0,\ldots,P-2\}$ beginnt seine Rechnung nach $P-k-1$ Sende- und $P-k-2$ Empfangsoperationen. Danach führt jeder Prozessor $k \in \{1,\ldots,P-2\}$ in dieser Phase $k+1$ Sende- und Empfangsoperationen aus, d.h. $l^{(k)}_{1,P-1} = 2\,(P-k) - 3 + 2\,(k+1) = 2\,P-1$. Prozessor 0 führt lediglich eine Empfangsoperation aus. Insgesamt,

$$l^{(k)}_{1,P-1} = r^{(k)}_{1,P-1} = \begin{cases} P-1 & \text{falls} \quad k=0 \\ 2\,P-1 & \text{falls} \quad 0<k<P-1 \\ 2\,P-2 & \text{falls} \quad k=P-1\ . \end{cases}$$

Phase 2: Jeder Prozessor $k \in \{1,\ldots,P-2\}$ berechnet zunächst die ersten $k-1$ Knoten in denen die Zonenbreite genau einen Knoten umfasst. Da für jeden Knoten gesendet und empfangen werden muss, gilt $l^{(k)}_{2,k-2} = l^{(k)}_{1,P-1} + 2\,(k-1)$. Betrachten wir nun einen Prozessor $k \in 1,\ldots,P-3$. Für den Eintrag in Stufe $k-1$ ist lediglich eine Empfangsoperation nötig. Prozessor k wäre deshalb zum Zeitpunkt $l^{(k)}_{2,k-1} = 2\,(P+k) - 2$ bereit, den rechten Eintrag in Stufe k zu berechnen. Hierfür wird das Ergebnis der Stufe $k-1$ von Prozessor $k+1$ benötigt. Dieses ist jedoch erst zum Zeitpunkt $l^{(k+1)}_{2,k-1} = 2\,(P+k) - 1$ versendet worden. Prozessor k beendet die Bearbeitung des Knotens in Stufe k somit zum Zeitpunkt $l^{(k)}_{2,k} = 2\,(P+k) + 1$. In Stufe k fällt noch eine Sendeoperation und in den verbleibenden $P-k-1$ Stufen fallen jeweils eine Sende- und eine Empfangsoperation an, also $l^{(k)}_{2,P-1} = 2\,(P+k) + 2 + 2\,(P-k-1) = 4\,P$. Wenn wir außerdem die Wartezeit von Prozessor 0 in Stufe 0 und die von Prozessor $P-1$ in Stufe $P-1$

berücksichtigen, erhalten wir

$$l_{2,P-1}^{(k)} = r_{2,P-1}^{(k)} + 1 = \begin{cases} 3\,P & \text{falls} \quad k = 0 \\ 4\,P & \text{falls} \quad 0 < k < P-2 \\ 4\,P-1 & \text{falls} \quad k = P-2 \\ 4\,P-3 & \text{falls} \quad k = P-1 \end{cases} .$$

Phase 3: Ähnlich wie in Phase 2 kommt es auch in dieser Phase zu Wartezeiten, wenn sich die Zonen verbreitern. Allerdings nehmen die Wartezeiten in Phase 3 mit abnehmendem Prozessorindex zu. Sei $k \in \{1, \ldots, P-3\}$. Dann beträgt die Wartezeit von Prozessor k genau $P-k-3$ Zeiteinheiten. Zusammen mit den erforderlichen $2\,P-1$ Kommunikationsoperationen erhalten wir die Berechnungszeit $l_{3,P-1}^{(k)} = 4\,P + P - k - 3 + 2\,P - 1 = 7\,P - k - 4$. Insbesondere gilt für die Stufe $P-2$ von Prozessor 1 somit $l_{3,P-2}^{(1)} = 7\,P - 6$. Prozessor 0 beendet seine Rechnung, nachdem er das Ergebnis dieser Stufe empfangen hat, d.h. $l_{3,P-1}^{(0)} = 7\,P - 5$. Bei Prozessor $P-2$ kommt es zu keinen Wartezeiten, also $l_{3,P-1}^{(0)} = 7\,P - 4$. Schließlich erhalten wir für Prozessor $P-1$ durch Berücksichtigung der Wartezeit in der ersten Stufe von Phase 3, $l_{3,P-1}^{(0)} = 5\,P - 2$. □

LEMMA 3.3.13. *Eine Pyramide der Höhe $n = s\,P$ mit $s \geq 3$ werde durch Ebenenplanung auf $P \geq 2$ Prozessoren abgebildet. Wenn die Berechnungszeit sowie die Latenzzeit vernachlässigt werden und der Kommunikationsoverhead $o = 1$ ist, dann ist die Berechnungsdauer gegeben durch*

$$T(n,\,P) = \begin{cases} 3\,s-1 & \textit{falls} \quad P = 2 \\ (2\,s+1)\,P-5 & \textit{falls} \quad P \geq 3 \end{cases} \leq (2\,s+1)\,P - 4 .$$

BEWEIS. Wir betrachten zunächst den Fall $P = 2$. In diesem Fall besteht jede Phase aus zwei Stufen. Mit Ausnahme der ersten Phase erfordert die erste Stufe von einem der beiden Prozessoren (Prozessor A) eine Empfangsoperation. Desweiteren ist in jedem Fall eine Sendeoperation notwendig. Die zweite Stufe erfordert jeweils von dem anderen Prozessor (Prozessor B) eine Empfangsoperation. Falls weitere Phasen folgen, kann Prozessor A gleichzeitig eine Sendeoperation durchführen. Insgesamt tragen also in der ersten Phase zwei und in allen weiteren Phasen drei Kommunikationsoperationen zur Laufzeit bei.

Für den Fall $P > 2$ erfolgt der Beweis durch Induktion nach s. Für $s = 3$ ist die Behauptung wegen Lemma 3.3.12 richtig, da $T(n,\,P) = 7\,P - 5 = (2 \cdot 3 + 1)\,P - 5$. Es sei nun $k \in \{1, \ldots, P-2\}$ und $s > 3$.

FALL 1. s gerade. Da $l_{s-1,P-1}^{(k)} > l_{s-1,P-1}^{(k+1)}$, kommt es an der rechten Grenze in den Stufen $0, \ldots, k$ zu keinen Wartezeiten bei Prozessor k. Bevor Prozessor k den linken Eintrag auf Stufe k berechnet, hat er 2 Kommunikationsoperationen mehr durchgeführt als Prozessor $k-1$ in den Stufen $0, \ldots, k-1$. Daher gilt $r_{s,k}^{(k)} > l_{s,k-1}^{(k-1)}$. Der linke Knoten in Stufe k von Prozessor k kann also ohne Wartezeit berechnet werden. In den Stufen $k+1, \ldots, P-1$ führen die Prozessoren k und $k-1$ die gleiche Anzahl an Kommunikationen aus. Es kommt also auch in diesen Stufen zu keinen Wartezeiten mehr.

FALL 2. s ungerade. Wegen $r_{s,0}^{(k)} = l_{s-1,P-1}^{(k)} + 1 > r_{s-1,P-1}^{(k-1)}$ können die ersten $P-k-1$ Stufen ohne Wartezeiten von Prozessor k berechnet werden. Am Ende von Stufe $P-k-2$ gilt $l_{s,P-k-2}^{(k)} = l_{s,P-k-2}^{(k+1)}$. Somit kann die Stufe $P-k-1$ ebenfalls ohne Wartezeit berechnet werden. Da der linke Knoten von dieser Stufe eine Sende- und eine Empfangsoperation erfordert,

vergrößert sich der Zeitunterschied an der rechten Grenze um eine weitere Zeiteinheit, so dass auch auf den restlichen Stufen keine Wartezeiten mehr entstehen.

Somit ist gezeigt, dass es bei den Prozessoren $1, \ldots, P-2$ zu keinen Wartezeiten in Phase s kommt. Da Prozessor 1 die 3. Phase als letzter beendet, beendet er folglich die folgenden Phasen als letzter. Er beendet bei der Berechnung einer Pyramide der Tiefe $s-1$ den letzten Knotens in Phase $s-1$ nach I.V. zum Zeitpunkt $(2s-1)\,P-5$. Bei einer Pyramide der Tiefe s endet die Bearbeitung dieses Knotens eine Zeiteinheit später, da eine zusätzliche Kommunikation zum Prozessor 0, falls s gerade ist, bzw. zum Prozessor 2, falls s ungerade ist, durchgeführt wird. Es folgen weitere $2\,P-1$ Kommunikationsoperationen, so dass $T(n,\,P) = (2s-1)\,P-5+1+2\,P-1 = (2s+1)\,P-5$. □

THEOREM 3.3.14. *Es sei $s,\ P \in \mathbb{N}$ mit $s \geq 3$ und es sei $n = s\,P$. Es sei*

$$T_{calc}(n,\,P) = \begin{cases} \frac{1}{2}\,\left((s^2+2)\,P+s-2\right)\,t & \textit{falls} \quad s \textit{ gerade} \\ \frac{1}{2}\,\left((s^2+1)\,P+s-1\right)\,t & \textit{falls} \quad s \textit{ ungerade}. \end{cases}$$

Für den Makespan $T(n,\,P)$ einer mit Pie-Mapping und Ebenenplanung organisierten Berechnung einer Pyramide der Höhe n auf P Prozessoren gilt

$$\begin{gathered} T_{calc}(n,\,P) + (s\,P-1)\,o + (P-1)\,L \\ \leq \quad T(n,\,P) \quad \leq \\ T_{calc}(n,\,P) + ((2\,s+1)\,P-4)\;o + (s\,P-1)\,L\ . \end{gathered}$$

BEWEIS. Die untere Schranke folgt aus der Tatsache, dass jeder Prozessor mindestens $s\,P-1$ Kommunikationsoperationen durchführt und dass mindestens die Latenzzeiten in den ersten P Stufen entstehen. Die obere Schranke folgt aus den Lemmata 3.3.1 und 3.3.13. □

Für den Overhead erhalten wir bei ungeradem s

$$T_{\mathrm{o}}(n,\,P) \leq \frac{1}{2}\,\left(P^2-P\right)\,t + \left(\left((2\,s+1)\,P^2-4\,P\right)\right)\,o + \left(sP^2-P\right)\,L\ .$$

Durch die Ebenenplanung wird somit der Einfluss des Kommunikationsoverheads auf den Gesamtoverhead um den Faktor s im Vergleich zum Pipelining- und ZS-Mapping verringert. Eine weitere Eigenschaft dieses Verfahrens ist, dass sich die Anzahl der Knoten, die ein Prozessor zwischen zwei Kommunikationen verarbeiten kann, in jeder Phase um eins erhöht. Nach etwa L/t Phasen beträgt der Rechenaufwand die Größenordnung der Latenzzeit, so dass sie durch die Rechnung versteckt werden kann. Die Grenze, ab der die Latenzzeit versteckt werden kann, kann jedoch noch verkleinert werden, wenn man die Anzahl der Knoten, die zwischen zwei Kommunikationsschritten berechnet werden, erhöht. Beispielsweise können wir die Abarbeitungsreihenfolge so verändern, dass innerhalb der Zonen nicht strikt von rechts nach links, sondern auch in die Tiefe gerechnet wird. Im nächsten Abschnitt betrachten wir eine Reihenfolge, die diesen Ansatz verfolgt.

3.3.5. Blockplanung. Bei der *Blockplanung* werden die Zonen in Blöcke unterteilt, die nacheinander berechnet werden. Abbildung 3.3.9 zeigt eine Blockplanung für 4 Prozessoren und $s = 9$. Alle Definitionen und Sätze in diesem Abschnitt beziehen sich auf diese gedrehte Form der Pyramidendarstellung, bei der die Knoten in einem kartesischem Koordinatensystem dargestellt werden. Die Indizierung der Knoten ändert sich dabei nicht. Wenn man einen Prozessor k mit $0 < k < P-1$ betrachtet, kann man sich die Konstruktion der s Blöcke in der Zone von Prozessor k wie folgt vorstellen. Jedes waagerechte Stück der unteren Zonengrenze bildet die untere

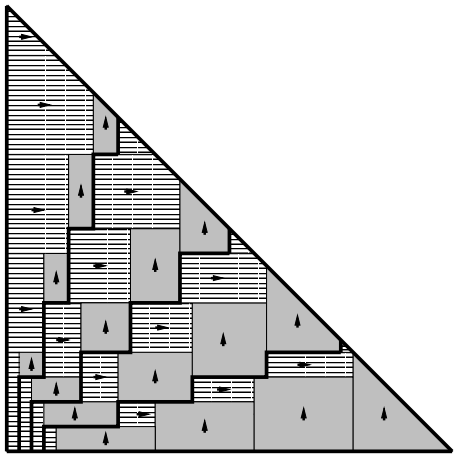


ABBILDUNG 3.3.9. Bei der Blockplanung werden die Zonen des Pie-Mapping Schemas in Blöcke eingeteilt, die nacheinander berechnet werden. Die hier gezeigte Abbildung entspricht einer um $135°$ gedrehten Pyramide. Die schraffierten Bereiche sind die ungeraden und die grauen Bereiche die geraden Blöcke. Die Pfeile geben die Auswertrichtung der Knoten innerhalb der Blöcke an.

Kante eines Blocks mit geradem Index und jedes senkrechte Stück der oberen Zonengrenze die linke Kante eines Blocks mit ungeradem Index.

Im Folgenden betrachten wir die Blockplanung einer Pyramide der Höhe $n = s\,P$ auf P Prozessoren.

DEFINITION 3.3.15. Es sei $k \in \{0, \ldots, P-1\}$ ein Prozessor. Bei der Blockplanung einer Pyramide der Höhe $s\,P$ wird die Zone von Prozessor k in s Blöcke $(1,\,k), \ldots, (s,\,k)$ eingeteilt. Wir nennen einen Block $(i,\,k)$ *gerade,* falls i gerade ist, und sonst *ungerade.* Ein Block ist *unvollständig,* wenn er von der Pyramidengrenze geschnitten wird. Mit $\sharp(i,\,k)$ bezeichnen wir die Anzahl der Knoten im Block $(i,\,k)$. Die Blöcke sind wie folgt definiert.

Blockindex i	Untere linke Ecke		Breite	Höhe
	x	y		
gerade	$(i-1)\,(P-k)$	$k\,i$	$2\,(P-k)$	i
ungerade	$i\,(P-k-1)$	$(i-1)\,(k+1)$	i	$2\,(k+1)$

Die Knoten in den ungeraden Blöcken werden zeilenweise von links nach rechts und die in den geraden Blöcken spaltenweise von unten nach oben berechnet.

Um die Wirkungsweise dieses Verfahrens zu verstehen, betrachten wir die Bearbeitung eines vollständigen geraden Blockes $(i,\,k)$ mit $k > 0$. Die einzigen Knoten, die auf Daten eines anderen Prozessors angewiesen sind, sind die in der unteren Zeile des Blocks. Da die Bearbeitung gerader Blöcke spaltenweise vorgenommen wird, und jede Spalte i Einträge hat, können somit i Einträge zwischen zwei Empfangsoperationen vorgenommen werden. Der Produzent dieser Daten ist Prozessor $k-1$. Ein Teil dieser Daten entsteht durch die Berechnung eines ungeraden Blocks, und (eventuell) ein anderer Teil durch die Berechnung eines geraden Blocks gleicher Höhe. Da der ungerade Block zeilenweise berechnet wurde, wurden die zu übermittelnden Daten unmittelbar nacheinander generiert. Somit driften die Zeitpunkte, zu denen die Daten verschickt,

und die, zu denen sie benötigt werden, immer weiter auseinander. Während der Bearbeitung der Daten, die von dem geraden Block stammen, bleibt die Zeitdifferenz konstant.

Allerdings setzt sich diese Tendenz nicht bis zum Schluss der Rechnung fort. Da gegen Ende die Blöcke nicht mehr vollständig sind, schrumpft die Zeitdifferenz zwischen Senden und Empfangen wieder zusammen. Um diesen Effekt analysieren zu können, ist eine genauere Analyse der Schnittmuster notwendig. Zunächst fragen wir uns nach einem Kriterium, mit dem wir beurteilen können, ob ein Block vollständig ist, oder nicht.

LEMMA 3.3.16. *Es sei $k \in \{0, \ldots, P-1\}$ ein Prozessor. Ein gerader Block $(i,\, k)$ ist vollständig, genau dann wenn*

$$i \leq \frac{(s-1)\,P + k + 1}{P+1} \ .$$

Ein ungerader Block $(i,\, k)$ ist vollständig, genau dann wenn

$$i \leq \frac{s\,P - k}{P+1} \ .$$

BEWEIS. Der obere rechte Endpunkt $(x,\, y)$ eines Blockes $(i,\, k)$ ist gegeben durch

Blockindex i	x	y
gerade	$(i+1)\,(P-k)-1$	$i\,(k+1)-1$
ungerade	$i\,(P-k)-1$	$(i+1)\,(k+1)-1$

Ein Punkt $(x,\, y)$ ist genau dann innerhalb der Pyramide, wenn er zu einer Ebene aus der Menge $\{0, \ldots, s\,P-1\}$ gehört. Also genau dann, wenn $x + y \leq P\,s - 1$. Wir erhalten die Behauptung des Lemmas, wenn wir diese Beziehung auf die Eckpunkte anwenden. □

LEMMA 3.3.17. *Falls ein Block $(i,\, k)$ mit $i > 1$ vollständig ist, dann ist auch der Block $(i-1,\, k)$ vollständig. Umgekehrt gilt, falls ein Block $(i,\, k)$ mit $i < s$ unvollständig ist, dann ist auch der Block $(i+1,\, k)$ unvollständig.*

BEWEIS. Es sei $(i,\, k)$ ein vollständiger gerader Block mit $i > 1$. Dann gilt

$$i \leq \frac{(s-1)\,P + k + 1}{P+1} \ \Rightarrow\ i - 1 \leq \frac{s\,P - 2\,P + k}{P+1} < \frac{s\,P - k}{P+1} \ .$$

Die restlichen Beziehungen lassen sich ebenfalls mit Hilfe von Lemma 3.3.16 einfach herleiten. □

Ein Schnitt durch einen Block schneidet jeweils zwei seiner Seiten. Je nachdem, welche Seiten geschnitten werden, unterscheiden wir verschiedene Schnittmuster. Hierfür bezeichnen wir die Seiten eines Blockes mit den Buchstaben *O*, *U*, *R* und *L* (für die obere, untere, linke und die rechte Seite). Um eine eindeutige Klassifizierung der Schnitte zu erhalten, ordnen wir den rechten unteren Eckpunkt der rechten Kante und den oberen linken Eckpunkt der oberen Kante zu. Ein Schnitt, der zuerst die obere Seite und dann die rechte Seite schneidet, bezeichnen wir dementsprechend als *OR*-Schnitt. Da der Schnitt von oben links nach unten rechts verläuft, sind die möglichen Schnittmuster *OR*, *LR*, *LU* und *OU*. Die folgenden beiden Lemmata setzen den Verlauf des Schnittes und die Lage sowie die Größe des Blockes in Verbindung.

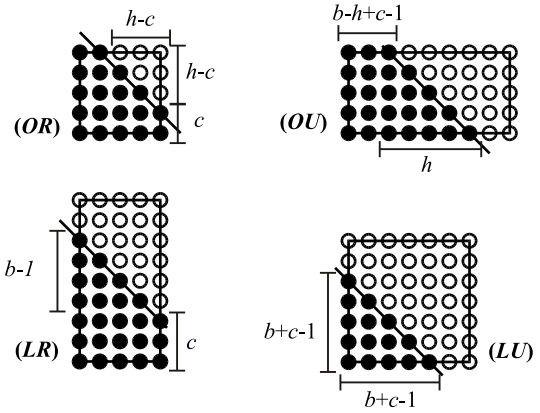


ABBILDUNG 3.3.10. Berechnung der Anzahl der Knoten in einem unvollständigen Block (Beweis von Lemma 3.3.19).

LEMMA 3.3.18. *Es sei $(i,\, k)$ ein unvollständiger Block der Breite b und der Höhe h. Die Position der unteren linken Ecke sei $(x,\, y)$. Es sei $c := P\, s - b + 1 - (x + y)$. Dann gelten folgende Äquivalenzen.*

Blockindex i	*gerade*	*ungerade*
$c \geq 1 \;\Leftrightarrow$	$i \leq \frac{(s-1)\,P+k}{P}$	$i \leq \frac{s\,P+k+1}{P+1}$
$b - h + c \geq 1 \;\Leftrightarrow$	$i \leq \frac{(s+1)\,P-k}{P+1}$	$i \leq \frac{s\,P-k-1}{P}$

BEWEIS. Die Aussagen folgen direkt aus der Definition 3.3.15 der Blockplanung. □

LEMMA 3.3.19. *Gegeben sei ein unvollständiger Block der Breite b und der Höhe h. Die Position der unteren linken Ecke sei $(x,\, y)$. Es sei $c := P\, s - b + 1 - (x + y)$. Dann wird die rechte Kante genau dann geschnitten wenn $c \geq 1$ gilt. Die obere Kante wird genau dann geschnitten wenn $b - h + c \geq 1$ gilt. Für die Anzahl der Elemente im Block gilt:*

	$c \geq 1$	$c < 1$
$b - h + c \geq 1$	$h\,b - \frac{1}{2}\,(h - c)\,(h - c + 1)$	$h\,(b - h + c - 1) + \frac{1}{2}\,h\,(h + 1)$
$b - h + c < 1$	$b\,c + \frac{1}{2}\,b\,(b - 1)$	$\frac{1}{2}\,(b + c)\,(b + c - 1)$

BEWEIS. Die Anzahl der Knoten, die auf der unteren Kante oder auf der rechten Kante liegen und noch innerhalb der Pyramide sind, ist gegeben durch $d := P\, s - (x + y)$. Es sei nun $c \geq 1$. Dann ist $d \geq b$ und deswegen wird die untere Kante nicht geschnitten. In diesem Fall bezeichnet c die Anzahl der Knoten auf der rechten Kante, die noch zur Pyramide gehören. Falls $b - h + c \geq 1$ gilt, ist $d \geq h$, was wiederum impliziert, dass die rechte Kante nicht geschnitten wird. Ferner ist in diesem Fall $b - h + c$ gerade gleich der Anzahl der Knoten auf der oberen Kante, die noch zur Pyramide gehören.

Die Anzahl der Elemente in dem Block ergibt sich damit gemäß Abbildung 3.3.10. □

Mit Hilfe von Lemma 3.3.18 und 3.3.19 können wir nun die Schnittform eines unvollständigen Blocks bestimmen. Tabelle 3.3.1 zeigt eine Zusammenfassung der möglichen Schnittmuster.

TABELLE 3.3.1. Schnittmuster eines unvollständigen Blocks $(i,\ k)$. Die eingeklammerten Muster kommen nach den Aussagen von Lemma 3.3.20 für ungerades s nicht vor.

i gerade	$i \leq \frac{(s-1)\,P+k}{P}$	$i > \frac{(s-1)\,P+k}{P}$
$i \leq \frac{(s+1)\,P-k}{P+1}$	OR	(OU)
$i > \frac{(s+1)\,P-k}{P+1}$	LR	(LU)

i ungerade	$i \leq \frac{s\,P+k+1}{P+1}$	$i > \frac{s\,P+k+1}{P+1}$
$i \leq \frac{s\,P-k-1}{P}$	OR	OU
$i > \frac{s\,P-k-1}{P}$	(LR)	LU

Es können durchaus mehrere Blöcke in einer Zone unvollständig sein. Beispielsweise liegen in dem in Abbildung 3.3.9 gezeigten Blockplanung die folgenden Schnittmuster vor.

Prozessor	Block			
k	$(6,\ k)$	$(7,\ k)$	$(8,\ k)$	$(9,\ k)$
0	vollst.	vollst.	OR	LU
1	vollst.	OR	LR	LU
2	vollst.	OR	LR	LU
3	vollst.	OR	LR	LU

Allerdings sind nicht alle Kombinationen von Schnittmustern möglich. Im Folgenden betrachten wir die möglichen Musterkombinationen für ungerades s genauer.

LEMMA 3.3.20. *Es sei s ungerade und es sei $k \in \{0, \ldots, P-1\}$ ein Prozessor. Es sei $(i_k,\ k)$ der erste unvollständige Block in der Zone von Prozessor k, und es sei $(i,\ k)$ ein beliebiger unvollständiger Block mit $i < s$. Dann gelten die folgenden Aussagen.*

(1) *Falls i gerade ist, wird der Block $(i,\ k)$ in der Form OR oder LR geschnitten. Sonst wird er in der Form OR oder OU geschnitten.*
(2) *Der Block $(i_k,\ k)$ wird genau dann in der Form LR geschnitten, falls i_k gerade ist und*

$$i_k = \frac{(s+1)\,P-k+1}{P+1}\ .$$

(3) *Falls i_k ungerade ist, wird der Block $(i_k,\ k)$ genau dann in der Form OU geschnitten, wenn*

$$i_k = \frac{s\,P+k+2}{P+1}\ .$$

(4) *Falls i gerade ist und $i > i_k$ gilt, wird der Block $(i,\ k)$ in der Form LR geschnitten.*
(5) *Falls i ungerade ist und $s > i > i_k$ gilt, wird der Block $(i,\ k)$ in der Form OU geschnitten.*
(6) *Der Block $(s,\ k)$ wird, falls $s > k+1$ gilt, in in der Form LU und sonst in der Form LR geschnitten. Im letzteren Fall ist $(s,\ k)$ der einzige unvollständige Block in der Zone.*

BEWEIS. (1): Da die Pyramidengrenze nur senkrechte Abschnitte der Zonengrenzen schneidet, können per Konstruktion der Blockplanung keine unteren Kanten von geraden Blöcken und keine linken Kanten von ungeraden Blöcken $(i,\, k)$ mit $i < s$ geschnitten werden.

(2) Der Block $(i_k,\, k)$ wird gemäß Tabelle 3.3.1 genau dann in der Form LR geschnitten, wenn $i_k > ((s+1)\,P - k)/(P+1)$ gilt. Da der Block $(i_k - 1,\, k)$ nach Voraussetzung vollständig ist, gilt

$$i_k - 1 \leq \frac{s\,P - k}{P+1} \text{ und damit } i_k \leq \frac{(s+1)\,P - k + 1}{P+1} \ .$$

Der Block $(i_k,\, k)$ wird folglich genau dann in der Form LR geschnitten, wenn $i_k = ((s+1)\,P - k + 1)/(P+1)$.

(3) Der Block $(i_k,\, k)$ wird genau dann in der Form OU geschnitten, wenn $i_k > (s\,P + k + 1)/(P+1) \geq 1$ gilt. Nun folgt wiederum die Behauptung aus der Tatsache, dass wegen der Vollständigkeit von Block $(i_k - 1,\, k)$ gleichzeitig $i_k \leq (s\,P + k + 2)/(P+1)$ gilt.

(4) Es sei $(i,\, k)$ ein gerader unvollständiger Block mit $i_k < i$. Dann ist der Block $(i-1,\, k)$ ein ungerader unvollständiger Block und somit gilt

$$i - 1 > \frac{Ps - k}{P+1} \Rightarrow i > \frac{(s+1)\,P - k + 1}{P+1} > \frac{(s+1)\,P - k}{P+1} \ .$$

(5) Nun sei $(i,\, k)$ ein ungerader unvollständiger Block mit $i_k < i < s$. In diesem Fall ist der Block $(i-1,\, k)$ ein gerader unvollständiger Block, und somit

$$i - 1 > \frac{(s-1)\,P + k + 1}{P+1} \Rightarrow i > \frac{s\,P + k + 2}{P+1} > \frac{s\,P + k + 1}{P+1} \ .$$

(6) Da $s > \frac{s\,P - k - 1}{P}$ gilt, wird stets die linke Seite von Block $(s,\, k)$ geschnitten. Die zweite Schnittseite ergibt sich aus

$$s \leq \frac{s\,P + k + 1}{P+1} \Leftrightarrow s \leq k + 1 \ .$$

Falls die rechte Seite des letzten Blocks geschnitten wird, ist der vorletzte Block vollständig, da

$$s \leq \frac{s\,P + k + 1}{P+1} \Rightarrow s \leq \frac{s\,P + k + 2}{P+1} \Leftrightarrow s - 1 \leq \frac{(s-1)\,P + k + 1}{P+1} \ .$$

□

Wenn wir die jeweils ersten unvollständigen Blöcke in den Zonen betrachten, ergeben sich weitere Eigenschaften sowohl der Indizes als auch der Schnittmuster.

LEMMA 3.3.21. *Es sei s ungerade und es sei i_k der Index des ersten unvollständigen Blocks in der Zone k. Es gelten folgende Eigenschaften.*

(1) *Es sei $k > 0$. Falls i_k gerade ist, dann ist $i_{k-1} = i_k$ und der Block $(i_{k-1},\, k-1)$ wird in der Form OR geschnitten.*

(2) *Es sei $0 \leq k < P - 1$. Falls i_k gerade ist, dann gilt $i_k - 1 \leq i_{k+1} \leq i_k + 1$.*
   (a) *Falls $i_{k+1} = i_k - 1$, dann wird der Block $(i_k,\, k)$ in der Form LR und der Block $(i_{k+1},\, k+1)$ in der Form OR geschnitten.*
   (b) *Falls $i_{k+1} = i_k + 1$, dann wird der Block $(i_k,\, k)$ in der Form OR und der Block $(i_{k+1},\, k+1)$ in der Form OU geschnitten.*

(3) *Es sei $0 \leq k < P - 1$. Falls i_k ungerade ist, dann ist $i_{k+1} = i_k$ und der Block $(i_{k+1},\, k+1)$ wird in der Form OR geschnitten.*

BEWEIS. (1.) Es sei $k > 0$ und es sei i_k gerade. Da $\frac{S\,P-k-1}{P+1} < \frac{s\,P-k}{P+1} < i_k$ gilt, ist der Block $(i_k,\, k-1)$ unvollständig (Lemma 3.3.16). Block $(i_k - 1,\, k-1)$ ist vollständig, da $i_k - 1 \leq \frac{s\,P-k}{P+1} < \frac{s\,P-(k-1)}{P+1}$. Wegen Lemma 3.3.17 sind auch alle Blöcke $(j,\, k-1)$ mit $j < i_k - 1$ vollständig und es gilt $i_{k-1} = i_k$.

Falls $i_k = 1$ gilt, wird der Block $(i_{k-1},\, k-1)$ in der Form OR geschnitten, da $i_k = 1 \leq \frac{(s+1)\,P-(k-1)}{P+1}$. Sonst ist der Block $(i_k - 1,\, k)$ vollständig, und es gilt $i_k - 1 \leq \frac{s\,P-k}{P+1}$, woraus wiederum $i_k \leq \frac{(s+1)\,P-(k-1)}{P+1}$ folgt.

(2.) Es sei $0 \leq k < P-1$ und es sei i_k gerade. Zunächst ist zu zeigen, dass der Block $(i_k - 2,\, k+1)$ vollständig ist falls $i_k > 2$. Es sei also $i_k > 2$. Da $(i_k - 1,\, k)$ vollständig ist, gilt $i_k - 1 \leq \frac{s\,P-k}{P+1}$ und damit auch $i_k - 2 \leq \frac{(s-1)\,P-k-1}{P+1} < \frac{(s-1)\,P+(k-1)+1}{P+1}$. Die Unvollständigkeit des Blocks $(i_k + 1,\, k+1)$ für $i_k < s$ folgt auf ähnliche Weise direkt aus der Tatsache, dass der Block $(i_k,\, k)$ unvollständig ist.

(2a.) Da der Block $(i_k - 1,\, k+1)$ unvollständig und der Block $(i_k - 1,\, k)$ vollständig ist, gilt

$$\begin{aligned} & \frac{s\,P-k-1}{P+1} && < i_k - 1 \leq && \frac{s\,P-k}{P+1} \\ \Leftrightarrow\ & \frac{(s+1)\,P-k}{P+1} && < i_k \leq && \frac{(s+1)\,P-k+1}{P+1}\ . \end{aligned}$$

Wegen der Ganzzahligkeit von i_k folgt hieraus direkt $i_k = (P\,(s+1) - k + 1)/(P+1)$. Wegen Lemma 3.3.20(2) wird der Block $(i_k,\, k)$ also in der Form LR geschnitten. Aus

$$i_{k+1} = i_k - 1 = \frac{s\,P-k}{P+1} < \frac{s\,P+(k+1)+1}{P+1}$$

folgt, dass der Block $(i_{k+1},\, k+1)$ in der Form OR geschnitten wird.

(2b.) Da der Block $(i_k,\, k+1)$ vollständig ist, gilt $i_k \leq \frac{s\,P-P+k+2}{P+1}$. Wegen $0 \leq k \leq P-2$ gilt $-P+k+2 \leq P-k$ und deswegen $i_k \leq \frac{s\,P+P-k}{P+1}$. Der Block $(i_k,\, k)$ wird also in der Form OR geschnitten. Wegen der Unvollständigkeit des Blocks $(i_k,\, k)$ gilt $i_k > \frac{(s-1)\,P+k+1}{P+1}$ und damit $i_k + 1 > \frac{s\,P+(k+1)+1}{P+1}$. Der Block $(i_k + 1,\, k+1)$ wird somit in der Form OU geschnitten.

(3.) Die Vollständigkeit von Block $(i_k - 1,\, k+1)$ folgt aus der Vollständigkeit von Block $(i_k - 1,\, k)$ und die Unvollständigkeit von Block $(i_k,\, k+1)$ folgt aus der Unvollständigkeit von Block $(i_k,\, k)$. Falls $i_k = 1$, gilt $i_k \leq \frac{s\,P+k+2}{P+1}$. Sonst folgt diese Ungleichung direkt aus der Vollständigkeit des Blockes $(i_k - 1,\, k)$. Der Block $(i_k,\, k+1)$ wird folglich in der Form OR geschnitten. □

Die Lemmata 3.3.20 und 3.3.21 reduzieren die möglichen Schnittmuster auf die beiden in Abbildung 3.3.11 dargestellten Szenarien. Die in Abbildung 3.3.9 dargestellte Blockplanung entspricht demnach Szenario 1.

Um das Laufzeitverhalten der Blockplanung zu analysieren, definieren wir analog zu Definition 3.3.9 eine Akkumulationsmatrix. Allerdings beziehen sich die Einträge in diesem Fall nicht auf die Ebenen der Pyramide, sondern auf die Blöcke.

ABBILDUNG 3.3.11. Die möglichen Schnittmuster lassen sich je nachdem, ob der Fall (2a) oder (2b) von Lemma 3.3.21 eintritt, in zwei Szenarien einteilen. Dargestellt sind jeweils 7 aufeinanderfolgende Blöcke und deren Schnittformen. Der erste Block ist jeweils ein ungerader Block.

DEFINITION 3.3.22. Die *Akkumulationsmatrix* $A(P,\, s) = (a_{k,i})\ \begin{matrix} k = 0, \ldots, P-1 \\ i = 0, \ldots, s \end{matrix}$ ist definiert durch

$$a_{k,i} = P - k - 1 + \sum_{j=1}^{i} \sharp(j,\, k) \quad .$$

Die Akkumulationsmatrix kann für konkrete Werte von P und s mittels der Definition 3.3.15 und durch Ausnutzen der Aussagen von Lemma 3.3.16 und 3.3.19 berechnet werden. Wir interessieren uns jedoch für geschlossene Formeln für die Einträge dieser Matrix. Um diese angeben zu können betrachten wir die verschiedenen Schnittmuster einzeln.

LEMMA 3.3.23. *Es sei $(i,\, k)$ ein vollständiger Block und es sei $A(P,\, k) = (a_{k,i})$ die Akkumulationsmatrix. Dann gilt*

$$a_{k,i} = \begin{cases} \frac{1}{2}(P+1)\, i^2 + (P-k)\, i + P - k - 1 & \textit{falls} \quad i \textit{ gerade} \\ \frac{1}{2}(P+1)\,(i^2+1) + (k+1)\, i - 1 & \textit{falls} \quad i \textit{ ungerade}. \end{cases}$$

BEWEIS. Summierung über die Blockgrößen gerader und ungerader Blöcke. □

Auf ähnliche Weise können wir auch geschlossene Formeln für unvollständige Blöcke $(i,\, k)$ angeben. Wir müssen dabei unterscheiden, ob der erste unvollständige Block $(i_k,\, k)$ gerade oder ungerade ist, in welcher Form er geschnitten wird, und ob i gerade oder ungerade ist. Insgesamt sind somit 8 Fälle zu unterscheiden. In jedem dieser Fälle sei der Wert von a_{ik} durch eine Funktion $a^{(\alpha|\beta|\gamma)}(i,\, k)$ mit $\alpha,\, \gamma \in \{\text{g, u}\}$ und $\beta \in \{\text{OR, OU, LR, LU}\}$ gegeben. Hierbei bezieht sich α auf den Block $(i_k,\, k)$ (*g* für gerade und *u* für ungerade), β auf das Schnittmuster von Block $(i_k,\, k)$ und γ auf den Block $(i,\, k)$. Betrachten wir zum Beispiel den Fall, dass der Block $(i_k,\, k)$ gerade ist und in der Form LR geschnitten wird. Dann gilt für jeden geraden Block $(i,\, k)$ mit

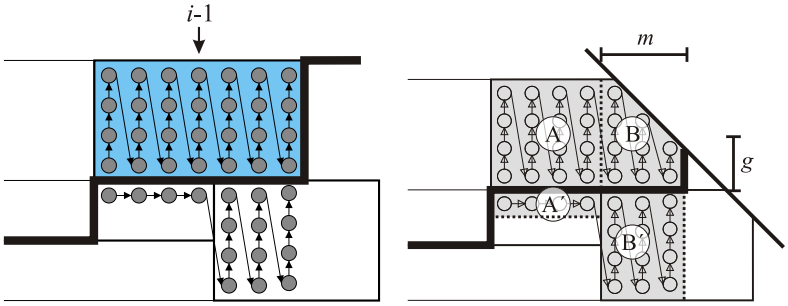


ABBILDUNG 3.3.12. Die Abarbeitung eines Blocks verläuft blockierungsfrei, wenn alle Blöcke ohne Wartezeit begonnen werden können.

$$i_k \le i < s,$$

$$\begin{aligned} a_{k,i} = a^{(\mathrm{g|LR|g})}(k,\, i_k,\, i) = \quad & \left(\frac{1}{2}i_k^2 - (k+1)\,i_k + (2+i)\,k - i - \frac{1}{2}i^2\right)\, P^2 \\ + \quad & \left((i - i_k - 2s)\,k + i_k^2 + (s - \frac{3}{2})\,i_k + \frac{1}{2}(i - i^2) + i\,s + 2\right)\, P \\ + \quad & (i_k - 2)\,s + \frac{1}{2}(i_k^2 - i_k + i) - 1 \ . \end{aligned}$$

Im Folgenden betrachten wir den Fall, dass alle Knoten die Berechnungszeit 1 haben und die Kommunikationskosten vernachlässigt werden können. Wir zeigen, dass der Eintrag $a_{k,i}$ der Akkumulationsmatrix die Zeit ist, zu der die Berechnung des Bock $(i,\, k)$ von Prozessor k beendet wird. Die Blockplanung führt somit in dieser Situation zu der gleichen Berechnungszeit wie die Ebenenplanung. Um dieses zu beweisen, müssen wir sicherstellen, dass kein Prozessor während der Berechnung auf ein Ergebnis warten muss. Wir nutzen hierfür die folgende Eigenschaft aus.

LEMMA 3.3.24. *Falls bei der Blockplanung jeder Block $(i,\, k)$ mit $i > 1$ ohne Wartezeit begonnen werden kann, dann braucht kein Prozessor auf Daten zu warten, sobald er mit der Berechnung des ersten Blocks begonnen hat.*

BEWEIS. Wir betrachten zunächst nur gerade Blöcke. Der Beweis verläuft durch Induktion nach k. Wenn Prozessor 0 einen geraden Block ohne Wartezeit beginnen kann, dann kann er ihn auch ohne Verzögerung zu Ende rechnen, da die Berechnung von keinen Daten abhängt, die von anderen Prozessoren produziert werden.

Nun betrachten wir einen geraden Block $(i,\, k)$ eines Prozessors $k > 0$ (vgl. Abbildung 3.3.12). Der Block $(i-1,\, k-1)$ hat die Breite $i-1$. Der erste Knoten in der obersten Zeile dieses Blocks wird früh genug berechnet, da der Block $(i,\, k)$ ohne Wartezeit begonnen werden kann. Da die Berechnung der letzten $i-2$ Knoten in dieser Zeile nur noch von bereits berechneten Daten abhängt, können sie ebenfalls früh genug zur Verfügung gestellt werden. Falls der Block $(i,\, k)$ mehr als $i-1$ Spalten hat, benötigen die letzten $m := 2\,(P-k) - (i-1)$ Spalten Daten von Block $(i,\, k-1)$. Der Block $(i,\, k-1)$ wird nach Induktionsvoraussetzung ohne Wartezeiten berechnet. Wir unterscheiden zwei Fälle.

FALL 1. Der Block $(i,\, k)$ ist vollständig. Da er die gleiche Höhe wie der Block $(i,\, k-1)$ hat, und die $i-1$-te Spalte von Block $(i,\, k)$ nicht vor der ersten Spalte von Block $(i,\, k-$

$1)$ begonnen werden kann, werden die restlichen m Spalten von Block $(i,\, k)$ ebenfalls ohne Wartezeit berechnet (vgl. Abbildung 3.3.12(a)).

FALL 2. Der Block $(i,\, k)$ ist unvollständig. Gemäß Abbildung 3.3.12(b) sei A die Menge der Knoten in den ersten $i-1$ Spalten von Block $(i,\, k)$ und B die Menge der Knoten in den restlichen Spalten. Ferner sei A' die Menge der Knoten in der obersten Zeile von Block $(i-1,\, k-1)$ und B' sei die Menge der Knoten in den ersten m Spalten von Block $(i,\, k-1)$. Es sei g die Höhe der letzten Spalte von Block $(i,\, k)$. Per Konstruktion gilt $g \geq 2$.

Angenommen, es läge ein LR Schnitt von Block $(i,\, k)$ vor. Dann hat die erste Spalte des Blocks $g + 2(P-k) - 1$ Knoten. Da $m > 0$ gilt, folgt $2(P-k) > i-1$. Dieses impliziert wiederum, dass die erste Spalte des Blocks mehr als i Knoten haben müsste, was per Konstruktion nicht möglich ist (Definition 3.3.15). Somit impliziert $m > 0$ einen OR-Schnitt.

Es sei nun $t_{A'}$ die Zeit, zu der der erste Knoten von A' berechnet ist, t_A die Berechnungszeit des ersten Knotens von A, $t_{B'}$ sei die Zeit zu der der letzte Knoten der Menge B' berechnet ist und t_B sei die Berechnungszeit des ersten Knotens in der letzten Spalte der Menge B (vgl. Abbildung 3.3.12(b)). Die Zeitdifferenz $\Delta t := t_A - t_{A'}$ ist nach Voraussetzung positiv. Somit gilt $t_B \geq t_{A'} + \Delta t + |A| + |B| - g + 1$. Ferner gilt per Induktionsvoraussetzung $t_{B'} = t_{A'} + |A'| + |B'| - 1$. Die Zeitdifferenz zwischen der Berechnung eines Datums durch Prozessor $k-1$ und der seiner Benutzung durch Prozessor k verringert sich während der Berechnung der letzten m Spalten. Da sie in der letzten Spalte von Block $(i,\, k)$ ein lokales Minimum annimmt, reicht es, $t_B > t_{B'}$ zu zeigen. Diese Zeitdifferenz lässt sich wie folgt abschätzen:

$$\begin{aligned} t_B - t_{B'} &\geq \Delta t + |A| + |B| - g - (|A'| + |B'|) + 2 \\ &\geq |A| + |B| - g - (|A'| + |B'|) + 2 \\ &= i\,(m + i - 1) - \frac{1}{2}(i-g)\,(i-g+1) - (i - 1 + im) - g + 2 \\ &= i^2 - 5\,i + 2\,ig - g^2 + 4 =: f(i,\, g)\ . \end{aligned}$$

Es sei $i_0(g) := -g + 5/2 + 1/2\,\sqrt{8g^2 - 20g + 9}$. Dann ist $f(i,\, g) > 0$ für alle $i, g \in \mathbb{N}$ mit $g \geq 2$ und $i > i_0(g)$. Wie man leicht nachrechnet, gilt $g \geq i_0(g)$. Wegen des OR-Schnitts gilt $i > g$ und daher insbesondere $i > i_0(g)$ und $f(i,\, g) > 0$. Somit ist also in der Tat die Zeitdifferenz $t_B - t_{B'}$ positiv und die Berechnung des Blocks $(i,\, k)$ blockierungsfrei.

Der Beweis für die ungeraden Blöcke erfolgt analog durch Induktion nach $P - k$. Falls hier der Block $(i,\, k)$ unvollständig ist, liegt entweder ein LR- oder ein OR-Schnitt vor. Im ersten Fall gilt wegen Lemma 3.3.20, $i = s$. Wenn wir die Mengen A, B, A' und B' sowie die Zeiten t_A, t_B, $t_{A'}$ und $t_{B'}$ analog definieren, erhalten wir bei einem LR-Schnitt die Aussage

$$t_B - t_{B'} \geq \frac{1}{2}s^2 - \frac{3}{2}s + 1 \geq 0\ .$$

Bei einem OR-Schnitt erhalten wir wie bei den geraden Blöcken $t_B - t_{B'} \geq f(i,\, g) > 0$, wobei g die Länge der letzten Zeile in Block $(i,\, k)$ darstellt. □

Wenn wir die Kommunikationskosten vernachlässigen, haben wir somit zwei Bedingungen nachzuweisen.

(1) Für alle geraden Blöcke $(i,\, k)$: Der erste Knoten in der letzten Zeile des Blocks $(i-1,\, k-1)$ ist berechnet, bevor der erste Knoten des Blocks $(i,\, k)$ berechnet werden soll.

(2) Für alle ungeraden Blöcke $(i,\ k)$: Der erste Knoten in der letzten Spalte des Blocks $(i-1,\ k+1)$ ist berechnet, bevor der erste Knoten des Blocks $(i,\ k)$ berechnet werden soll.

DEFINITION 3.3.25. Es sei $(i,\ k)$ ein Block der Breite b und der Höhe h. Die Position des unteren linken Knotens sei gegeben durch $(x,\ y)$ und es sei $c := P\,s - b + 1 - (x+y)$. Dann sei

$$\ell(i,\ k) := \begin{cases} (2\,(P-k)-1)\ i+1 & \text{falls} \quad (i,\ k) \text{ gerade und vollständig,} \\ \sharp(i,\ k) - c + 1 & \text{falls} \quad (i,\ k) \text{ gerade und unvollständig,} \\ (2\,k+1)\,i+1 & \text{falls} \quad (i,\ k) \text{ ungerade und vollstständig,} \\ \sharp(i,\ k) - b + h - c + 1 & \text{falls} \quad (i,\ k) \text{ ungerade und unvollständig.} \end{cases}$$

LEMMA 3.3.26. *Es sei $1 \le i < s$. Ein gerader Block $(i,\ k)$ enthält $\ell(i,\ k)-1$ Knoten in den ersten $2\,(P-k)-1$ Spalten. Ein ungerader Block $(i,\ k)$ enthält $\ell(i,\ k)-1$ Knoten in den ersten $2\,(k+1)-1$ Zeilen.*

THEOREM 3.3.27. *Es sei $s \ge 3$ ungerade. Es sei $A(P,\ s) = (a_{k,i})$ die Akkumulationsmatrix und es sei $(i,\ k)$ ein Block mit $i > 1$. Falls i gerade ist und $0 < k \le P-1$, dann gilt*

$$(a_{k,i-1}+1) - (a_{k-1,i-2} + \ell(i-1,\ k-1)) =$$
$$\begin{cases} 2\,i-2 > 0 & \textit{falls } (i-1,\ k-1) \textit{ vollst. und } (i-1,\ k) \textit{ vollst.} \\ \frac{2\,(s-2)\,P+2\,(P-k)+P+1}{P+1} > 0 & \textit{falls } (i-1,\ k-1) \textit{ vollst. und } (i-1,\ k) \textit{ unvollst.} \\ 2\ ((s-i+1)\,P+1-k)) - 1 > 0 & \textit{sonst.} \end{cases}$$

Falls i ungerade ist und $0 \le k < P-1$, dann gilt

$$(a_{k,i-1}+1) - (a_{k+1,i-2} + \ell(i-1,\ k+1)) =$$
$$\begin{cases} 2\,i-1 > 0 & \textit{falls } (i-1,\ k+1) \textit{ vollst. und } (i-1,\ k) \textit{ vollst.} \\ \frac{(2\,s-1)\,P+2\,k+3}{P+1} > 0 & \textit{falls } (i-1,\ k+1) \textit{ vollst. und } (i-1,\ k) \textit{ unvollst.} \\ 2\ ((s-i)\,P+2+k)) - 1 > 0 & \textit{sonst.} \end{cases}$$

BEWEIS. Für den Beweis sind die sich aus Lemma 3.3.21 ergebenden Kombinationen von Schnittmustern zweier benachbarter Prozessoren zu berücksichtigen. Es sei also $(i,\ k)$ ein gerader Block mit $i > 1$ und $0 < k \le P-1$.

FALL 1. Der Block $(i-1,\ k-1)$ ist vollständig. Für $i > 2$ ist gemäß Lemma 3.3.17 dann auch der Block $(i-2,\ k-1)$ vollständig. Der Block $(i-1,\ k)$ könnte jedoch unvollständig sein.

FALL 1.1. Der Block $(i-1,\ k)$ ist vollständig. In diesem Fall sind die Werte $a_{i-1,\,k}$ und $a_{i-2,\,k-1}$ durch Lemma 3.3.23 gegeben. Es gilt:

$$\underbrace{\frac{1}{2}(P+1)\,((i-1)^2+1) + (k+1)\,(i-1)}_{a_{i-1,\ k+1}+1} -$$
$$\left(\underbrace{\frac{1}{2}(P+1)\,(i-2)^2 + (P-k+1)\,(i-2) + P - k}_{a_{i-2,\ k-1}} + \underbrace{(2k-1)\,(i-1)+1}_{\ell(i-1,\,k-1)} \right) \quad = \quad 2i-2\ .$$

FALL 1.2. Der Block $(i-1,\ k)$ ist unvollständig. Hier wird gemäß Lemma 3.3.21(2a) der Block $(i-1,\ k)$ in der Form OR und der Block $(i,\ k-1)$ in der Form LR geschnitten. Wir haben

also den folgenden Ausdruck zu betrachten.

$$\begin{aligned} & a^{(\mathrm{u}|\mathrm{OR}|\mathrm{u})}(k,\, i-1,\, i-1) + 1 - (a_{k+1,i-2} + \ell(i-1,\, k+1)) = \\ & \qquad \left(-\frac{1}{2}\left(s^2+i^2+1\right) + (i-1)\,s + i\right)\, P^2 \\ & + \quad \left(\frac{1}{2}\left(3\,i-s-1\right) + (k+i)\;s + (1-i)\;k - i^2\right)\, P \\ & + \quad \frac{1}{2}\left(5\,i - i^2 - k^2 + k\right) - k\,i - 2 \quad . \end{aligned} \tag{3.3.1}$$

Wegen 3.3.20(2) gilt

$$i = i_{k-1} = \frac{P\,(s+1) - k + 2}{P+1} \quad . \tag{3.3.2}$$

Die Behauptung des Theroms folgt durch Einsetzen von (3.3.2) in (3.3.1).

FALL 2. Der Block $(i-1,\, k-1)$ ist unvollständig. Wegen Lemma 3.3.21(3) ist dann auch der Block $(i-1,\, k)$ unvollständig. Je nachdem ob der erste unvollständige Block von Prozessor $k-1$ gerade oder ungerade ist, unterscheiden wir die folgenden Fälle.

FALL 2.1. Der erste unvollständige Block von Prozessor $k-1$ ist gerade. Wegen Lemma 3.3.21(2) gilt dann $i_{k-1}-1 \le i_k \le i_{k-1}+1$, und folglich sind drei weitere Fälle zu unterscheiden.

FALL 2.1.1. $i_k = i_{k-1} - 1$. Hier wird der Block $(i_k,\, k)$ in der Form OR und der Block $(i_{k-1},\, k-1)$ in der Form LR geschnitten. Wir betrachten also

$$\begin{aligned} & a^{(\mathrm{u}|\mathrm{OR}|\mathrm{u})}(k,\, i_k,\, i-1) + 1 - (a^{(\mathrm{g}|\mathrm{LR}|\mathrm{g})}(k-1,\, i_k+1,\, i-2) + \ell(i-1,\, k+1)) = \\ & \qquad -\frac{1}{2}(s-i_k)^2\, P^2 \\ & + \quad \left(\frac{1}{2}(3\,s+i_k) + (i_k+k)\,s - i_k^2 - 2\,i - k\,i_k + 2\right)\, P \\ & + \quad \frac{1}{2}\left(i_k - i_k^2 - 2\,k i_k - k^2 + 1\right) \quad . \end{aligned} \tag{3.3.3}$$

Gemäß 3.3.20(2) gilt

$$i_k = i_{k-1} - 1 = \frac{P\,s - k + 1}{P+1} \quad . \tag{3.3.4}$$

Nun folgt die Behauptung von Einsetzen von (3.3.4) in (3.3.3).

FALL 2.1.2. $i_k = i_{k-1}$. Nun ist i_k gerade und folglich wird der Block $(i_{k-1},\, k-1)$ wegen 3.3.21(1) in der Form OR geschnitten. Es sind somit die beiden möglichen Schnittmuster des Blocks $(i_k,\, k)$ zu betrachten.

FALL 2.1.2.1. Der Block $(i_k,\, k)$ wird in der Form OR geschnitten. Hier folgt die Behauptung direkt aus der Auswertung von

$$a^{(\mathrm{g}|\mathrm{OR}|\mathrm{u})}(k,\, i_k,\, i-1) + 1 - \left(a^{(\mathrm{g}|\mathrm{OR}|\mathrm{g})}(k-1,\, i_k,\, i-2) + \ell(i-1,\, k+1)\right) \quad .$$

Fall 2.1.2.2. Der Block $(i_k,\, k)$ wird in der Form LR geschnitten. In diesem Fall gilt wegen Lemma 3.3.20(2)

$$i_k = \frac{P\,(s+1) - k + 1}{P+1} \quad ,$$

und die Behauptung folgt durch Einsetzen dieser Eigenschaft in

$$a^{(\mathrm{g|LR|u})}(k,\, i_k,\, i-1) + 1 - \Big(a^{(\mathrm{g|OR|g})}(k-1,\, i_k,\, i-2) + \ell(i-1,\, k+1)\Big) \quad .$$

FALL 2.1.3. $i_k = i_{k-1} + 1$. Hier werden wegen Lemma 3.3.21(2b) die Blöcke $(i_{k-1},\, k-1)$ und $(i_k,\, k)$ in den Formen OR beziehungsweise OU geschnitten, und es gilt wegen Lemma 3.3.20(3) die Beziehung

$$i_k = i_{k-1} + 1 = \frac{s\,P + k + 2}{P+1} \quad . \tag{3.3.5}$$

Insgesamt folgt die Behauptung durch Einsetzen von (3.3.5) in

$$a^{(\mathrm{u|OU|u})}(k,\, i_k,\, i-1) + 1 - \Big(a^{(\mathrm{g|OR|g})}(k-1,\, i_k-1,\, i-2) + \ell(i-1,\, k+1)\Big) \quad .$$

Fall 2.2. Der erste unvollständige Block von Prozessor $k-1$ ist ungerade. Wegen Lemma 3.3.21(3) gilt in diesem Fall $i_k = i_{k-1}$ und der Block $(i_k,\, k)$ wird in der Form OR geschnitten. Wir haben nun die beiden möglichen Schnittformen des Blocks $(i_k,\, k-1)$ zu betrachten.

FALL 2.2.1. Der Block $(i_k,\, k-1)$ wird ebenfalls in der Form OR geschnitten. Dann folgt die Behauptung direkt durch Auswertung von

$$a^{(\mathrm{u|OR|u})}(k,\, i_k,\, i-1) + 1 - \Big(a^{(\mathrm{u|OR|g})}(k-1,\, i_k,\, i-2) + \ell(i-1,\, k+1)\Big) \quad .$$

FALL 2.2.2. Der Block $(i_k,\, k-1)$ wird in der Form OU geschnitten. Es gilt dann wegen Lemma 3.3.20(3)

$$i_k = i_{k-1} = \frac{s\,P + k + 1}{P+1} \quad ,$$

womit die Behauptung aus

$$a^{(\mathrm{u|OR|u})}(k,\, i_k,\, i-1) + 1 - \Big(a^{(\mathrm{u|OU|g})}(k-1,\, i_k,\, i-2) + \ell(i-1,\, k+1)\Big)$$

folgt.

Der Beweis des Theorems für einen ungeraden Block $(i,\, k)$ erfolgt analog. Hier werden für den Fall, dass der Block $(i-1,\, k+1)$ vollständig ist, die Fälle, in denen der Block $(i-1,\, k)$ vollständig bzw. nicht vollständig ist, unterschieden. Falls der Block $(i-1,\, k+1)$ unvollständig ist, wird die weitere Fallunterscheidung danach vorgenommen, ob der erste unvollständige Block von Prozessor k gerade oder ungerade ist. □

Es sei s eine ungerade natürliche Zahl. Eine Pyramide der Höhe $s\,P$ werde durch Blockplanung auf P Prozessoren abgebildet. Es sei $A(P,\, s) = (a_{k,i})$ die Akkumulationsmatrix. Wenn alle Knoten die Berechnungszeit 1 haben und die Kommunikationskosten vernachlässigt werden, dann beendet Prozessor k den Block $(i,\, k)$ zum Zeitpunkt $a_{k,i}$. Insbesondere beenden alle Prozessoren die letzte Ebene zum Zeitpunkt $((s^2+1)P + s - 1)/2$.

BEWEIS. Gemäß Theorem 3.3.27 ist die Blockplanung blockierungsfrei. Folglich ergeben sich die selben Berechnungszeiten wie bei der Ebenenplanung. □

Da bei der Blockplanung dieselben Kommunikationsvorgänge auftreten wie bei der Ebenenplanung, wirkt sich der Kommunikationsoverhead in gleicher Weise aus. Auf Grund des anders verlaufenden kritischen Pfades erhalten wir einen etwas geringeren Kommunikationsoverhead.

LEMMA 3.3.28. *Eine Pyramide der Höhe $n = s\,P$ mit $s \geq 3$ werde durch Blockplanung auf $P \geq 2$ Prozessoren abgebildet. Wenn die Berechnungszeit sowie die Latenzzeit vernachlässigt werden und der Kommunikationsoverhead $o = 1$ ist, dann ist die Berechnungsdauer gegeben durch*

$$T(n,\,P) = \begin{cases} 3\,s - 1 & \textit{falls} \quad P = 2 \\ 2\,s\,P - 1 & \textit{falls} \quad P \geq 3 \end{cases} \leq 2\,s\,P - 1 \ .$$

BEWEIS. Der Beweis für den Fall $P = 2$ erfolgt analog zum Beweis von Lemma 3.3.13 für die Ebenenplanung. Für $P \geq 3$ betrachten wir den Verlauf des kritischen Pfades. In Abbildung 3.3.13 ist sein Verlauf für sechs Prozessoren und $s = 5$ dargestellt.

Prozessor 2 beendet die Berechnung seines ersten Knotens zum Zeitpunkt $2\,(P-2)-1$. Die restlichen beiden Knoten in der ersten Phase und der erste Knoten in der zweiten Phase erfordern jeweils 2 Kommunikationen. Folglich kann Prozessor 1 den ersten Knoten in der zweiten Stufe von Phase 2 zum Zeitpunkt $2\,P + 1$ bearbeiten. Nach diesem Zeitpunkt entstehen bei Prozessor 1 keine weiteren Wartezeiten mehr. Da für jede der beiden Zonengrenzen noch $(s-1)\,P - 1$ Kommunikationen folgen, erhalten wir insgesamt eine Bearbeitungszeit von

$$T(n,\,P) = 2\,P + 1 + 2\,((s-1)\,P - 1) = 2\,sP - 1 \ .$$

□

THEOREM 3.3.29. *Es sei $s,\ P \in \mathbb{N}$ mit $s \geq 3$ und es sei $n = s\,P$. Es sei*

$$T_{calc}(n,\,P) = \begin{cases} \frac{1}{2}\,\left((s^2+2)\,P + s - 2\right)\,t & \textit{falls} \quad s \textit{ gerade} \\ \frac{1}{2}\,\left((s^2+1)\,P + s - 1\right)\,t & \textit{falls} \quad s \textit{ ungerade}. \end{cases}$$

Für den Makespan $T(n,\,P)$ einer mit Pie-Mapping und Blockplanung organisierten Berechnung einer Pyramide der Höhe n auf P Prozessoren gilt

$$\begin{gathered} T_{calc}(n,\,P) + (s\,P - 1)\,o + (P-1)\,L \\ \leq \quad T(n,\,P) \quad \leq \\ T_{calc}(n,\,P) + (2\,s\,P - 1)\ o + (s\,P - 1)\,L \ . \end{gathered}$$

BEWEIS. Die untere Schranke folgt aus der Tatsache, dass jeder Prozessor mindestens $s\,P-1$ Kommunikationsoperationen durchführt und dass mindestens die Latenzzeiten in den ersten P Stufen entstehen. Die obere Schranke folgt aus den Lemmata 3.3.1 und 3.3.28. □

Der entscheidende Unterschied der beiden Abbildungsverfahren wird an Theorem 3.3.27 deutlich. Die Zeit, die bei der Blockplanung für die Kommunikation zur Verfügung steht, steigt, solange die Blöcke vollständig sind, an. Wie in Abbildung 3.3.5 zu sehen ist, steht bei der Ebenenplanung entlang der nach links verlaufenden Zonengrenzen keine Zeit für die Kommunikation zur Verfügung. An diesen Stellen muss es also zwangläufig zu Verzögerungen kommen. Die Auswirkungen dieser Verzögerungen sind in dem in Abbildung 3.3.14(a) gezeigten Ausführungsprofil einer feingranularen Rechnung mit Ebenenplanung deutlich sichtbar.

Abbildung 3.3.14(b) zeigt die zur Verfügung stehende Kommunikationszeit bei Ebenen- und Blockplanung sowie die sich aus Theorem 3.3.27 ergebende untere Schranke für die Blockplanung. Mit Ausnahme der Ebenen, in denen die Boxen nicht vollständig sind, steht beim Blockmapping mehr Zeit für die Kommunikation zur Verfügung als bei Ebenenplanung. Der Beitrag der Latenzzeit zum Gesamtoverhead der Berechnung ist somit bei der Blockplanung geringer. Dieses wird durch Abbildung 3.3.14(c) veranschaulicht. Es existiert zwar bei beiden Verfahren eine Pyramidenhöhe, ab der der Einfluss der Latenzzeit konstant ist, jedoch tritt dieser Effekt

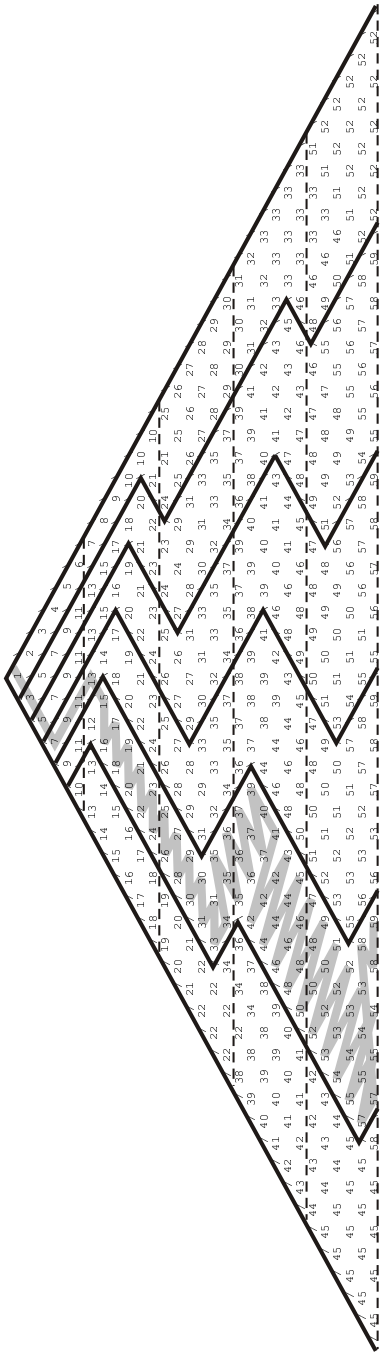
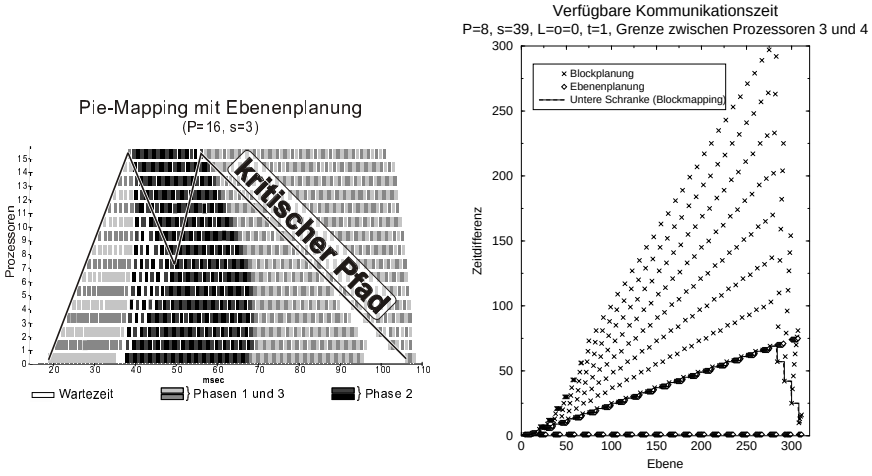
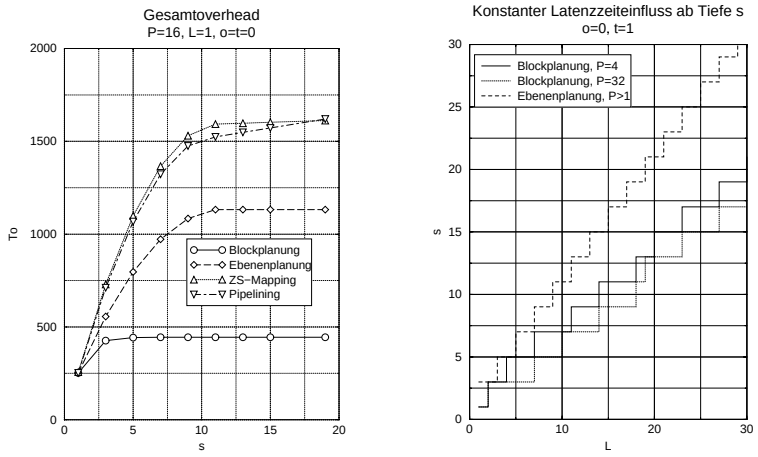


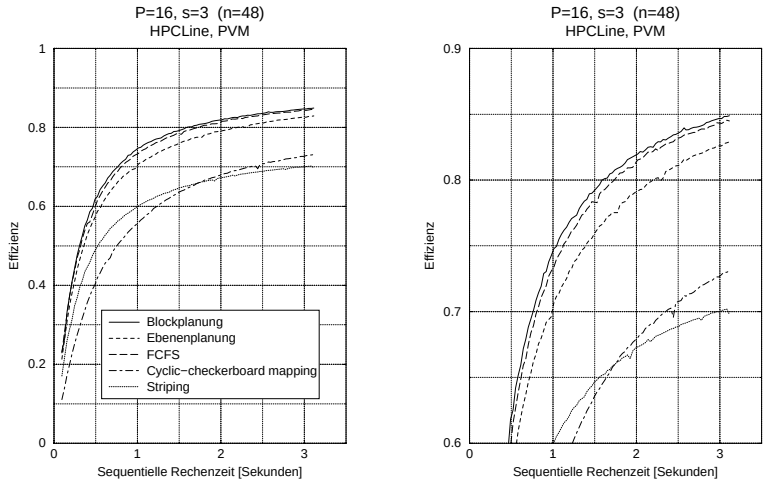
ABBILDUNG 3.3.13. Verlauf des kritischen Pfades bei Blockplanung für $P = 6$, $s = 5$, $L = t = 0$ und $o = 1$. Ab einschließlich Phase 3 entstehen durch den Kommunikionsoverhead nur noch Wartezeiten bei den Prozessoren 0 und $P - 1$, die jedoch (wie bei der Ebenenplanung) nicht auf dem kritischen Pfad liegen.



(a) (b)



(c) (b)



(e)

ABBILDUNG 3.3.14. Vergleich der Abbildungsverfahren für die Pyramide.

bei der Blockplanung bereits bei kleineren Pyramidenhöhen ein. Abbildung 3.3.14(d) stellt den minimalen Wert für s, ab dem der Einfluss konstant ist, für beide Verfahren da. Im Falle der Ebenenplanung hängt dieser Wert nicht von der Prozessoranzahl ab und ist ungefähr so groß wie die Latenzzeit. Bei der Blockplanung hängt dieser Wert zwar von der Prozessoranzahl ab, ist aber nur etwa halb so groß wie bei Ebenenplanung.

Die Blockplanung stellt einen deterministischen Ansatz dar, die durch die Ebenenplanung erzeugten Verzögerungen zu verhindern. Ein weiterer Ansatz dieses zu tun besteht darin, die Knoten innerhalb der Zonen in der Reihenfolge zu bearbeiten, in der sie berechenbar werden (FCFS-Reihenfolge).

Abbildung 3.3.14(e) zeigt einen experimentellen Vergleich der in diesem Abschnitt behandelten Abbildungsverfahren. Das Pie-Mapping erlaubt bei allen Abarbeitungsreihenfolgen wegen des nur linear in der Pyramidenhöhe ansteigenden Kommunikationsaufwands höhere Effizienzen als das ZS-Mapping und Pipelining. Die FCFS-Reihenfolge ist effizienter als Ebenenplanung, da Wartezeiten durch die Bearbeitung bereits berechenbarer Knoten überbrückt werden. Da die Blockplanung zudem die Berechnungsreihenfolgen der Prozessoren aufeinander abstimmt, führt es vor allem bei feingranularen Berechnungen zu höheren Effizienzen als die FCFS-Reihenfolge.

Das Diagramm zeigt zusätzlich einen Effekt, der bereits bei der Diskussion des ZS-Mappings und des Pipelinings aufgezeigt wurde. Für im Vergleich zur Rechenzeit große Kommunikationszeiten ist in der Tat das Pipelining effizienter als das ZS-Mapping.

3.4. Offene Fragen

Wir haben gesehen, dass der Kommunikationsaufwand reduziert werden kann, wenn den Prozessoren zusammenhängende Gebiete des Aufgabengraphen zugewiesen werden. Das Pie-Mapping Verfahren kann durch Spiegelungs- und Umkehrungsoperationen direkt für umgekehrte Pyramiden und für Diamanten angewendet werden. Die Anwendung auf den Diamanten wirft die Frage auf, ob andere Auswertungsreihenfolgen besser sind als die, die sich aus dem zusammengesetzten Plan einer Pyramide und einer umgekehrten Pyramide ergibt. Ferner ist es eine interessante Aufgabe, den Pie-Mapping Ansatz so zu verallgemeinern, dass er auf Rechtecke und auf höherdimensionale Indexräume angewendet werden kann. Ein Verfahren, das automatisch geeignete Gebietszerlegungen und Ausführungsreihenfolgen festlegt, könnte im Bereich der parallelisierenden Compiler sinnvoll eingesetzt werden.

Während der Berechnung der ersten P Ebenen einer Pyramide treten zwangsläufig Wartezeiten auf. Insbesondere diese Phase könnte also durch redundante Pläne beschleunigt werden.

Generell stellt sich schließlich die Frage nach der Komplexität, einen optimalen Plan für die Pyramide zu berechnen. Wie in Abschnitt 3.1 dargelegt, ist in der Literatur die Konstruktion von Plänen mit und ohne Aufgabenduplizierung für verschiedene Graphklassen betrachtet worden. Es ist bislang nicht bekannt, ob ein optimaler (redundanter) Plan für die Pyramide in polynomieller Zeit angegeben werden kann. Da die Pyramide durch ihre Höhe eindeutig spezifiziert ist, müsste die Laufzeit eines solchen Algorithmus polylogarithmisch in der Pyramidenhöhe sein.

KAPITEL 4

Planung parallelisierbarer Aufgaben

In diesem Kapitel betrachten wir Anwendungen, bei denen Parallelität in verschiedenen Ebenen auftritt. Derartige Anwendungen generieren *Aufgaben*, die ihrerseits parallel berechnet werden können. Vor allem wenn nur wenige Aufgaben gleichzeitig berechnet werden können, ist es notwendig, diese Aufgabenparallelität auszunutzen. Dieses geschieht im Allgemeinen am effizientesten, wenn den verschiedenen Aufgaben disjunkte Prozessorgruppen zugeordnet werden.

Die Zuordnung kann beispielsweise dadurch geschehen, dass allen Aufgaben initial die gleiche Prozessoranzahl zugeordnet wird. Sobald eine Aufgabe berechnet ist, werden die frei gewordenen Prozessoren anderen Aufgaben zugeteilt. Eine solche dynamische Methode wird von dem in VDS integrierten Konzept der *dynamischen Prozessorgruppen* unterstützt. Es ist zwar bei der Nutzung dieses Konzepts bereits möglich, durch eine Gewichtung der Aufgaben Vorabwissen über ihre Skalierbarkeit und Laufzeit zu nutzen, jedoch können bessere Ergebnisse erzielt werden, indem dieses Wissen für eine initiale Planung der Berechnung genutzt wird.

Im Folgenden betrachten wir das Problem einer solchen initialen Planung. Wir gehen dabei davon aus, dass die Aufgaben dieselbe parallele Rechenzeit $T(x,\ P)$ haben, und dass diese Rechenzeit a priori bekannt ist. Abschnitt 4.1 fasst den Stand der Forschung auf diesem Gebiet zusammen. In Abschnitt 4.2 stellen wir einen Algorithmus vor, der einen optimalen Plan für gleichförmige Aufgaben berechnet. Bei einer konstanten Prozessoranzahl ist die Laufzeit des Algorithmus polynomiell in der Aufgabenanzahl. Da die Laufzeit exponentiell in der Prozessoranzahl ist, stellen wir in Abschnitt 4.3 einen $2-$Approximationsalgorithmus für das Problem vor. Schließlich diskutiert Abschnitt 4.4 die Anwendung des Algorithmus für baumstrukturierte Berechnungen. Insbesondere werden zwei Aussagen über die Isoeffizienz solcher Berechnungen hergeleitet, die bei der Analyse der Skalierbarkeit der parallelen Descartes Methode in Kapitel 7 benutzt werden.

4.1. Stand der Forschung

Das Problem, die Bearbeitung paralleler Aufgaben zu planen, ist in den letzten 10 Jahren in unterschiedlichen Ausprägungen untersucht worden. Diese ergeben sich zum Beispiel dadurch, dass die Bearbeitung einer Aufgabe unterbrochen werden kann [**87**], oder durch die Existenz von Abhängigkeiten zwischen den Aufgaben [**132, 133**]. Weitere Ausprägungen ergeben sich durch unterschiedliche Optimierungsziele wie zum Beispiel der gewichteten Summe der Bearbeitungszeiten [**119, 121**] oder der gesamten Bearbeitungszeit. Veltman u.a. [**131**] und Drozdowski [**47**] geben gute Übersichten über die Vielzahl der Arbeiten.

In diesem Kapitel betrachten wir die unterbrechungsfreie Abarbeitung von n unabhängigen Aufgaben auf P Prozessoren. Das Ziel ist, die Gesamtbearbeitungszeit (Makespan) zu minimieren. Wenn die Anzahl der Prozessoren, mit denen eine Aufgabe berechnet werden kann, unterschiedliche Werte annehmen darf, werden die Aufgaben als *verformbar* bezeichnet. Bei nicht verformbaren Aufgaben ist somit die Prozessoranzahl jeder Aufgabe ein Teil der Eingabe.

Bei verformbaren Aufgaben enthält die Eingabe anstatt dessen Angaben über das Skalierungsverhalten der Aufgaben—die Bestimmung der Prozessoranzahl jeder Aufgabe ist hier ein Teil des Planungsproblems. Da das Planungsproblem sowohl für verformbare als auch für nicht verformbare Aufgaben selbst für eine feste Anzahl von Prozessoren ($P \geq 5$) im strengen Sinne NP-hart ist [**48**], stellt sich die natürliche Frage nach Approximationsalgorithmen.

Das Planungsproblem für nicht verformbare Aufgaben (*PNVA*) ist ein Spezialfall des Ressourcen-beschränkten Planungsproblems (*RBP*), das von Garey und Graham [**60**] formuliert wurde. Hierbei gibt es eine oder mehrere Ressourcen, von denen jede Aufgabe jeweils eine bestimmte Menge benötigt. Das PNVA kann als RBP mit einer Ressource, den Prozessoren, formuliert werden. Garey and Graham [**60**] zeigen, dass ein einfacher auf Listenplanung beruhender Ansatz ausreicht, um einen Approximationsfaktor von 2 zu erreichen.

Falls den Aufgaben nur Prozessoren mit aufeinanderfolgenden Indizes zugeordnet werden können, ist das PNVA mit dem orthogonal orientierten Packungsproblem für Rechtecke identisch, das zuerst von Baker, Coffman und Rivest untersucht wurde [**3**]. Die Dimensionen des Packungsproblems sind die Zeit und die Prozessoren. Der beste bekannte Approximationsfaktor 2 für dieses Problem ist von Steinberg vorgestellt worden [**122**]. Für eine eingeschränkte Form des PNVA, bei der für jede Aufgabe sogar die Prozessormenge spezifiziert wird, geben Amoura u.a. ein polynomielles Approximationsschema an, mit dem ein Approximationsfaktor von $1 + \varepsilon$ für $\varepsilon > 0$ erreicht werden kann [**1**]. Allerdings wird hierbei zum einen die Prozessoranzahl als konstant angenommen und zum anderen ist die Laufzeit exponentiell in $1/\varepsilon$.

Das Planungsproblem für verformbare Aufgaben (*PVA*) ist eine Verallgemeinerung des PNVA. Jedes PNVA kann als PVA formuliert werden, indem für das Skalierungsverhalten einer Aufgabe bei jeder Prozessoranzahl, die nicht der geforderten entspricht, ein "schlechter" Wert angegeben wird. Hierdurch wird sichergestellt, dass eine optimale Lösung des PVA jeder Aufgabe die geforderte Prozessoranzahl zuordnet.

Das PVA besteht aus zwei Teilproblemen. Die Erste besteht in der Ermittlung der Prozessormenge für jede Aufgabe und die Zweite in der Bestimmung der Zeitpunkte, zu denen jeweils die Bearbeitung der Aufgaben begonnen werden soll. Die Approximationsalgorithmen lassen sich danach klassifizieren, ob sie diese Teilprobleme nacheinander oder gleichzeitig lösen. Das PVA wurde zuerst von Krishnamurti und Ma [**86**] untersucht. Sie lösen beide Probleme gleichzeitig, allerdings mit der Einschränkung, dass alle Aufgaben zum Zeitpunkt 0 starten müssen. (Dieses beschränkt die Anzahl der Aufgaben auf die Prozessoranzahl.) Die verwendete Technik besteht darin, zunächst jeder Aufgabe einen Prozessor zuzuordnen. Danach wird iterativ jeweils der Aufgabe, die die längste Laufzeit hat, ein weiterer Prozessor zugeordnet—solange bis alle Prozessoren zugeteilt sind. Bei P Prozessoren und $n < P$ Aufgaben erreichen sie einen Approximationsfaktor von $\min(P,\, 2/(1 - n/P))$.

Der schnellste Approximationsalgorithmus mit Approximationsfaktor $2/(1 - 1/P)$ wurde von Belkhale und Banerjee vorgestellt [**5**]. Er stellt zwar keine Restriktionen an die Anzahl der Aufgaben, fordert aber für jede Aufgabe, dass bezüglich der Prozessoranzahl ihre Berechnungszeit monoton sinkt und ihre Arbeit monoton steigt. Auch hier werden die beiden Teilprobleme gleichzeitig verfolgt. Alle Aufgaben, denen mehr als ein Prozessor zugeordnet wird, starten zum Zeitpunkt 0. Die anderen Aufgaben werden mit Listenplanung zugeteilt. In jeder Iteration wird die Aufgabe J mit der längsten Laufzeit betrachtet. Falls diese zu denen gehört, die zuletzt fertig

werden, und noch nicht alle Prozessoren zugeordnet sind, wird J ein zusätzlicher Prozessor zugeteilt (was eine neue Planung der sequentiellen Aufgaben zur Folge hat). Andernfalls terminiert der Algorithmus. Die Berechnungzeit wird von $(n + P) \log P$ dominiert.

Turek, Wolf und Yu haben die erste Arbeit vorgestellt, in der die beiden Schritte (Prozessorzuteilung und Lösung des resultierenden PNVA) getrennt voneinander behandelt werden [**127**]. Sie zeigen, dass man einen Algorithmus für das PNVA mit Laufzeit $L(n,\ P)$ und Approximationsfaktor f benutzen kann, um einen Algorithmus für das PVA mit demselben Approximationsfaktor zu bekommen ohne Anforderungen an die Aufgaben zu stellen. Die Laufzeit des konstruierten Algorithmus wird von $Pn\ L(n,\ P)$ dominiert. Sie resultiert daraus, dass höchstens $n(P-1)+1$ mögliche Prozessorzuweisungen berechnet werden und für jede dieser Zuweisungen ein Plan bestimmt wird. Ludwig und Tiwari verbessern dieses Ergebnis bezüglich der Laufzeit, indem sie nur noch für eine Zuweisung einen Plan bestimmen [**95**]. Somit wird die Planungszeit bereits von $nP + L(n,\ P)$ dominiert. Zusammen mit dem Packungsalgorithmus von Steinberg erreichen somit beide Algorithmen einen Approximationsfaktor von 2.

Mounie, Rapine und Trystram stellen für den Fall, dass die Aufgaben keinen superlinearen Speedup ermöglichen, einen $\sqrt{3}$-Approximationsalgorithmus vor [**101**]. Sie benutzen einen dualen Approximationsansatz. Hierbei wird entweder

- ein Plan mit Makespan von höchstens λd erzeugt, oder
- es wird gezeigt, dass es keinen Plan mit Makespan d gibt.

Durch eine geeignete Wahl des Startwertes für d erhält man mit einer konstanten Anzahl von Bisektionierungsschritten eine hinreichend genaue Schätzung des Makespans eines optimalen Plans und somit eine Approximationgarantie von λ. Ausgehend von dem Schätzwert d wird für jede Aufgabe eine minimale Prozessoranzahl bestimmt, so dass ihre Bearbeitungszeit höchstens d Zeiteinheiten benötigt. Danach werden die Aufgaben in absteigender Reihenfolge der Bearbeitungszeit platziert. Falls die Gesamtarbeit der Aufgaben, die zum Zeitpunkt 0 beginnen, kleiner als $d\sqrt{3}/2m$ ist, kann für dieses Vorgehen bereits eine Approximationsgüte von $\sqrt{3}$ garantiert werden, und die Laufzeit des Algorithmus wird von $n \log P$ dominiert. Andernfalls wird das Planungsproblem auf ein Rucksackproblem zurückgeführt, was eine von $\min(n\ P,\ n^3 + n\ \log P)$ dominierte Laufzeit ergibt.

Das erste polynomielle Approximationsschemata für eine konstante Prozessoranzahl wurde von Jansen und Porkolab vorgestellt [**77**]. Sie benutzen wie Amoura u.a. eine LP Formulierung für das Problem. Eine erweiterte Variante, bei der für jede Aufgabe alternative Prozessormengen spezifiziert werden können wird von Chen und Miranda [**17**] betrachtet. Auch sie geben ein polynomielles Approximationsschema an.

In diesem Abschnitt betrachten wir eine spezielle Form (*GPVA*) des PVA, bei der alle Aufgaben dieselbe sequentielle Laufzeit und dasselbe Skalierungsverhalten haben. Es wird gezeigt, dass das Problem in diesem Fall einfacher wird, da bei konstanter Prozessoranzahl die Zeit für die Berechnung einer optimalen Lösung polynomiell in der Aufgabenanzahl ist. Allerdings ist die Eingabegröße in diesem Fall nur noch logarithmisch in der Anzahl der Aufgaben, so dass hierdurch die Frage nach der Komplexität des GPVA nicht beantwortet wird. Da die Prozessoranzahl exponentiell in die Laufzeit dieses Algorithmus eingeht, ist er für praktische Zwecke ungeeignet. Es wird daher ein Approximationsalgorithmus für das GPVA vorgestellt, der die Gleichförmigkeit der Aufgaben berücksichtigt.

4.2. Optimale Pläne für das GPVA

Formal stellt sich das GPVA wie folgt:

Eingabe: Eine Prozessoranzahl P, eine Aufgabenanzahl m, und eine monoton fallende Laufzeitfunktion $t : \{1, \ldots, P\} \to \mathbb{R}^{>0}$, wobei $t(p)$ die Bearbeitungszeit einer Aufgabe auf p Prozessoren ist.

Ausgabe: Ein *Plan* $S \subseteq \mathcal{P}\{1, \ldots, P\} \times \mathbb{R}^{\geq 0}$ mit

(1) $|S| = m$,

(2) $(\sigma_1, \tau_1), (\sigma_2, \tau_2) \in S$ und $\sigma_1 \cap \sigma_2 \neq \emptyset$ impliziert $[\tau_1, \tau_1 + t(|\sigma_1|)] \cap [\tau_2, \tau_2 + t(|\sigma_2|)] = \emptyset$,

(3) $\max\{T \mid (\sigma, \tau) \in S, T = \tau + t(|\sigma|)\}$ ist minimal.

Zunächst sei bemerkt, dass die Forderung nach der Monotonie der Laufzeitfunktion keine Einschränkung darstellt. Falls ein Problem vorliegt, dessen Laufzeitfunktion nicht monoton ist, reicht es aus, eine modifizierte Laufzeitfunktion $t^*(p) := \min_{1 \leq i \leq p} t(p)$, die gegebenenfalls Prozessoren ungenutzt lässt, zu betrachten.

Bezüglich der Ausgabespezifikation stellt die erste Bedingung sicher, dass für jede Aufgabe eine Prozessormenge und ein Startzeitpunkt berechnet wird, die zweite Bedingung impliziert, dass ein Prozessor höchstens an einer Aufgabe gleichzeitig arbeitet, und die dritte Bedingung enthält das Optimierungsziel—die Minimierung der Gesamtbearbeitungszeit. Ein Plan ist *zulässig*, wenn er die ersten beiden Bedingungen erfüllt. Er ist *optimal*, wenn er zusätzlich die dritte Bedingung erfüllt.

Wir werden im Folgenden zeigen, dass die Zeit, einen optimalen Plan zu finden, von $n\,(2n+1)^{P^{2-P}}\,2^P$ dominiert wird. Falls die Ausführungszeiten rational sind, wird sie bereits von $n\,(g\,t(1))^{P-1}\,2^P$ dominiert, wobei g das kleinste gemeinsame Vielfache der Ausführungszeiten ist. Um dieses zu zeigen, werden wir zunächst zeigen, dass es stets einen optimalen Plan gibt, der bestimmte Eigenschaften erfüllt. Für die Beschreibung dieser Eigenschaften werden die folgenden Definitionen benötigt.

DEFINITION 4.2.1. Der *Lastvektor* eines Plans S ist der Vektor $(\ell_1, \ldots, \ell_P)$ mit

$$\ell_i = \max\{\tau + t(|\sigma|) \mid (\sigma, \tau) \in S \wedge i \in \sigma\}$$

für $1 \leq i \leq P$. Der *sortierte Lastvektor* $(l_1, \ldots, l_P)$ ist gegeben durch $l_i = \ell_{\pi(i)}$ für $1 \leq i \leq P$, wobei π eine Permutation mit $\ell_{\pi(i)} \leq \ell_{\pi(j)}$ für $1 \leq i < j \leq P$ ist. Der Wert l_P ist der *Makespan* $M(S)$ des Plans S.

DEFINITION 4.2.2. Es sei $(l_1, \ldots\ l_P)$ der sortierte Lastvektor des Plans S. Der *normalisierte Lastvektor* $K(S) = (k_1, \ldots, k_{P-1})$ von S ist gegeben durch $k_i = l_{i+1} - l_1$ für $1 \leq i \leq P-1$.

DEFINITION 4.2.3. Ein Plan S heißt *gepackt*, wenn für jedes Paar $(\sigma, \tau) \in S$ entweder $\tau = 0$ gilt, oder wenn es ein anderes Paar $(\sigma', \tau') \in S$ gibt mit $\sigma \cap \sigma' \neq \emptyset$ und $\tau' + t(|\sigma'|) = \tau$.

DEFINITION 4.2.4. Es sei S ein zulässiger Plan für m Aufgaben. Der Plan S *enthält* einen Plan S' genau dann wenn $S' \subseteq S$. Die Pläne $S_1, \ldots, S_{m-1}$ heißen *Zwischenpläne von* S, wenn $S_1 \subseteq \cdots \subseteq S_{m-1} \subseteq S$, und $|S_i| = i$ für $1 \leq i < m$.

DEFINITION 4.2.5. Es sei S ein zulässiger Plan. Eine Aufgabe $(\sigma, \tau) \in S$ heißt *unabhängig*, wenn jede Aufgabe $(\sigma', \tau') \in S$ mit $\sigma \cap \sigma' \neq \emptyset$ die Bedingung $\tau' \geq \tau$ erfüllt.

Generell können wir einen Plan für $m > 1$ Aufgaben konstruieren, indem wir zunächst einen Plan für $m-1$ Aufgaben berechnen und dann der m-ten Aufgabe eine geeignete Menge freier

Prozessoren zuweisen. Jede derartige schrittweise Konstruktion eines Plans definiert eine Menge von Zwischenplänen.

Wir werden nun normalisierte Lastvektoren optimaler Pläne und entsprechender Zwischenpläne betrachten.

LEMMA 4.2.6. *Für jeden optimalen Plan S für m Aufgaben gibt es einen Plan S_m sowie Zwischenpläne $S_1, \ldots, S_{m-1}$ von S_m mit den folgenden Eigenschaften für $1 \leq i \leq m$:*

(1) *Der Plan S_m hat denselben Makespan wie der Plan S.*
(2) *Der Plan S_i ist gepackt und es gilt für den normalisierten Lastvektor $(k_1^{(i)}, \ldots, k_{P-1}^{(i)})$ von S_i die Ungleichung $k_{P-1}^{(i)} \leq t(1)$. Keine Aufgabe in S_i wird später beendet als in S.*

BEWEIS. Wir konstruieren die Pläne $S_1, \ldots, S_m$ iterativ. Die Mengen $R_1, \ldots, R_m$ enthalten die noch nicht geplanten Aufgabe. Während der Konstruktion gilt mit $S_0 = \emptyset$ und $R_0 = S$ die folgende Invariante für $1 \leq i \leq m$.

(1) $S_i \cup R_i$ ist ein zulässiger Plan für m Aufgaben.
(2) Die Aufgabe $(\sigma, \tau) \in S_i \setminus S_{i-1}$ ist in R_{i-1} unabhängig.

Für den Anfang wählen wir eine beliebige Aufgabe $(\sigma, \tau) \in S$ mit $\tau = 0$ und setzen $S_1 := \{(\sigma, \tau)\}$, $R_1 := S \setminus S_1$. Eine solche Aufgabe existiert, da S optimal ist. Wegen $\tau \leq t(1)$ erfüllt S_1 den zweiten Teil der Behauptung. Offensichtlich erfüllen S_1 und R_1 die Invariante. Es sei nun $1 < i \leq m$ und es gelte die Behauptung für die Pläne $S_1, \ldots S_{i-1}$.

Es sei $(l_1, \ldots, l_P)$ der sortierte Lastvektor von S_{i-1} und es sei π die zugehörige Sortierpermutation. Es sei f die Anzahl der Prozessoren, deren Last gleich l_1 ist. Somit gilt $l_1 = \cdots = l_f$. Wir unterscheiden die folgenden Fälle.

FALL 1. Es gibt eine unabhängige Aufgabe $(\sigma, \tau) \in R_{i-1}$, die einen der Prozessoren $\pi(1), \ldots, \pi(f)$ benutzt. Dann setzen wir $R_i := R_{i-1} \setminus (\sigma, \tau)$. Es sei $k \in \{1, \ldots, f\}$ so gewählt, dass $\pi(k) \in \sigma$. Wegen der zweiten Bedingung der Invariante gilt $\tau \geq l_1$.

FALL 1.1 $\tau + t(|\sigma|) - l_1 \leq t(1)$. Es sei τ^* der früheste Zeitpunkt, zu dem alle Prozessoren aus σ in S_{i-1} fertig sind. Da $S_{i-1} \cup R_{i-1}$ ein zulässiger Plan ist, gilt $\tau^* \leq \tau$. Wir setzen $S_i := S_{i-1} \cup \{(\sigma, \tau^*)\}$. Damit ist auch $S_i \cup R_i$ ein zulässiger Plan. Es sei $(l'_1, \ldots, l'_P)$ der sortierte Lastvektor von S_i. Da in S_i kein Prozessor früher fertig ist als in S_{i-1}, gilt $l_1 \leq l'_1$. Ferner gilt $l'_P = \max(l_P, \tau + t(|\sigma|))$, und folglich $l'_P - l'_1 \leq l'_P - l_1 \leq t(1)$.

FALL 1.2. $\tau + t(|\sigma|) - l_1 > t(1)$. Dann setzen wir $S_i := S_{i-1} \cup \{(\{\pi(k)\}, l_1)\}$. Durch diese Operation wird der Zeitpunkt, zu dem die Aufgabe (σ, τ) berechnet ist, verkürzt und folglich ist auch in diesem Fall $S_i \cup R_i$ ein zulässiger Plan. Für den sortierten Lastvektor $(l'_1, \ldots, l'_P)$ von S_i gilt $l_1 \leq l'_1$ und $l'_P = \max(l_P, l_1 + t(1))$. Somit gilt auch $l'_P - l'_1 \leq l'_P - l_1 \leq t(1)$.

FALL 2. Es gibt keine unabhängige Aufgabe aus R_{i-1}, die einen der Prozessoren $\pi(1)$, ..., $\pi(f)$ benutzt. Es sei (σ, τ) eine unabhängige Aufgabe aus R_{i-1}, so dass $\tau + t(|\sigma|)$ minimal ist. (Es gilt also für jede unabhängige Aufgabe $(\sigma', \tau') \in R_{i-1}$, die Ungleichung $\tau' + t(|\sigma'|) \geq \tau + t(|\sigma|)$.) Wir setzen $R_i := R_{i-1} \setminus (\sigma, \tau)$. Wie im ersten Fall ist wegen der zweiten Bedingung der Invariante $\tau \geq l_1$.

FALL 2.1. $\tau + t(|\sigma|) - l_1 \leq t(1)$. Wie in Fall 1.1 sei τ^* der früheste Zeitpunkt, zu dem alle Prozessoren aus σ in S_{i-1} fertig sind. Wir setzen $S_i := S_{i-1} \cup \{(\sigma, \tau^*)\}$. Es sei $(l'_1, \ldots, l'_P)$ der sortierte Lastvektor von S_i. Mit der Begründung aus Fall 1.1 gilt $\tau^* \leq \tau$ und $l'_P - l'_1 \leq t(1)$. Außerdem ist $S_i \cup R_i$ ein zulässiger Plan.

FALL 2.2. $\tau + t(|\sigma|) - l_1 > t(1)$. Da keine unabhängige Aufgabe aus R_{i-1} den Prozessor $\pi(1)$ benutzt, und keine Aufgabe vor dem Zeitpunkt $l_1 + t(1) < \tau + t(|\sigma|)$ fertig wird, können wir $S_i :=$

$S_{i-1} \cup \{(\{\pi(1)\},\ l_1)\}$ setzen und erhalten mit $S_i \cup R_i$ einen zulässigen Plan. Es sei $(l'_1, \ldots, l'_P)$ der sortierte Lastvektor von S_i. Wie in Fall 1.2 gilt $l_1 \leq l'_1$ und $l'_P = \max(l_P, l_1 + t(1))$. Schließlich folgt $l'_P - l'_1 \leq l'_P - l_1 \leq t(1)$.

In allen Fällen ist der resultierende Plan S_i gepackt und es ist $S_i \cup R_i$ zulässig. Außerdem wird in S_i keine Aufgabe später beendet als in S. Insbesondere ist also S_m ein optimaler Plan. □

Wir berechnen optimale Pläne indem wir iterativ Zwischenpläne einer optimalen Lösung konstruieren. Wegen Lemma 4.2.6 können wir die Suche auf gepackte Pläne mit normalisierten Lastvektoren $(k_1, \ldots, k_{P-1})$, für die $0 \leq k_{P-1} \leq t(1)$ gilt, beschränken. Wir nennen solche Pläne *relevant*. Um den Algorithmus formulieren zu können, führen wir den Begriff der *nachfolgenden Pläne* ein.

DEFINITION 4.2.7. Es sei $i \geq 0$ und es sei S ein relevanter Plan für i Aufgaben (die leere Menge ist ein Plan für 0 Aufgaben). Wenn S' ein relevanter Plan für $i+1$ Aufgaben ist, der S enthält, dann *folgt der Plan S' auf S*. Es sei C_n die Menge der relevanten Pläne für maximal n Aufgaben. Wir definieren die Relation "$\rightarrow$" auf dem Menge C_n indem $S \rightarrow S'$ genau dann gilt, wenn der Plan S' auf S folgt.

Algorithmus 3 Optimale Pläne

Eingabe: Ein GPVA mit n Aufgaben, P Prozessoren, $t : \{1, \ldots, P\} \rightarrow \mathbb{R}^{>0}$
Ausgabe: Ein optimaler Plan für das GPVA

```
1: Q ← {∅}
2: for i = 1 to n
3:     Q' ← {}
4:     for each a ∈ Q
5:         for each b ∈ C_n mit a → b
6:             if (∃ c ∈ Q' | K(c) = K(b)) then
7:                 if M(b) < M(c) then Q' ← Q' \ {c} ∪ {b}
8:             else
9:                 Q' ← Q' ∪ {b}
10:    Q ← Q'
11: return (arg min_{a∈Q} {M(a)})
```

THEOREM 4.2.8. *Es gibt einen Algorithmus für das GPVA, dessen Laufzeit von $n\,(2n+1)^{P^2-P}\,2^P$ dominiert wird.*

BEWEIS. Der Algorithmus 3 berechnet einen optimalen Plan mittels dynamischer Programmierung. Er speichert alle relevanten Pläne für i Aufgaben in der Menge Q. Sobald alle relevanten Pläne für i Aufgaben generiert worden sind, werden die Pläne für $i+1$ Aufgaben berechnet indem für jeden Plan in Q alle nachfolgenden Pläne generiert werden. Diese werden in Q' gespeichert. Wenn mehrere relevante Pläne denselben normalisierten Lastvektor haben, wird der Plan gespeichert, der den kleinsten Makespan hat (Zeilen 6-9).

Es sei S ein relevanter optimaler Plan und es seien $S_1, \ldots, S_{m-1}$ relevante Zwischenpläne von S. Solche Pläne existieren wegen Lemma 4.2.6. Für jeden dieser Pläne gibt es keinen anderen Plan mit demselben normalisierten Lastvektor und einem kleineren Makespan – sonst wäre S nicht optimal. Da S_2 dem Plan S_1 nachfolgt, S_3 dem Plan S_2 nachfolgt und so weiter, findet der Algorithmus diese Pläne und schließlich den optimalen Plan S.

Um die Laufzeit des Algorthmus zu bestimmen, betrachten wir die Anzahl der Schritte, die der Algorithmus in jeder Iteration der äußeren Schleife durchführt. Es sei $(l_1, \ldots, l_P)$ ein sortierter Lastvektor eines relevanten Plans. Da alle Pläne gepackt sind, fängt die Bearbeitung einer Aufgabe entweder zum Zeitpunkt 0 an oder direkt im Anschluss an die Bearbeitung einer anderen Aufgabe. Wir können deswegen die Einträge der Lastvektoren als Linearkombination der möglichen Berechnungszeiten einer Aufgabe schreiben. Für $1 \leq i \leq P$ erhalten wir

$$\begin{aligned} l_i &\in \{a_1 t(1) + \cdots + a_P t(P) \mid a_1, \ldots, a_P \in \{0, \ldots, n\}\} \\ \Rightarrow \quad k_i = l_{i+1} - l_1 &\in \{a_1 t(1) + \cdots + a_P t(P) \mid a_1, \ldots, a_P \in \{-n, \ldots, n\}\} \ . \end{aligned}$$

Es gibt also höchstens $(2n+1)^{P^2-P}$ unterschiedliche normalisierte Lastvektoren und somit Pläne in der Menge Q. Da es höchstens 2^P Möglichkeiten gibt, eine Aufgabe zu platzieren, und insgesamt n Iterationen durchgeführt werden, wird die Anzahl der Schritte von $n\,(2n+1)^{P^2-P}\,2^P$ dominiert. □

Es sei R ein GPVA mit einer rationalen Laufzeitfunktion $t : \{1, \ldots n\} \to \mathbb{Q}^{>0}$. Es sei $g = \text{kgV}\,(t(1), \ldots, t(P))$. Dann wird die Zeit für die Berechnung eines optimalen Plans für R von $n\ (g\,t(1))^{P-1}\ 2^P$ dominiert.

BEWEIS. Wir betrachten einen sortierten relevanten Lastvektor $(l_1, \ldots, l_P)$. Für $1 \leq i \leq P$ gilt

$$l_i \in \left\{ \frac{a}{g} \mid a \in \mathbb{N}_0 \right\} \ \Rightarrow\ k_i = l_{i+1} - l_1 \in \left\{ \frac{a}{g} \mid a \in \{0, \ldots, g\,t(1)\} \right\} \ ,$$

da $0 \leq k_i \leq t(1)$. Die Anzahl der relevanten Pläne wird somit von $(g\,t(1))^{P-1}$ dominiert. □

Gegeben sei ein GPVA mit P Prozessoren. Es sei $q \in \mathbb{N}$ mit $1 \leq q \leq P$. Wenn die Laufzeitfunktion durch

$$t(p) = \begin{cases} \frac{1}{p} & \text{für} \quad 1 \leq p \leq q \\ \frac{1}{q} & \text{für} \quad p > q \end{cases} \ .$$

gegeben ist, dann wird die Zeit, die benötigt wird, um einen optimalen Plan zu berechnen, von $n\,q!^{P-1}\ 2^P$ dominiert

BEWEIS. Das kleinste gemeinsame Vielfache der Ausführungszeiten ist durch $q!$ beschränkt. Damit folgt die Aussage aus Folgerung 4.2. □

Wir haben gezeigt, dass es möglich ist, einen optimalen Plan in linearer Zeit zu berechnen, wenn die Prozessorzahl konstant ist und die Berechnungszeiten rational sind. Allerdings hängt die Laufzeitfunktion exponentiell von der Prozessoranzahl ab. Somit ist das Verfahren für realisitische Systemgrößen nicht einsetzbar. Für praktische Zwecke werden im nächsten Abschnitt *phasenparallele Pläne* vorgestellt – ein Approximationsalgorithmus, der Pläne berechnet, die eine bestimmte Struktur aufweisen.

4.3. Phasenparallele Pläne

Im Folgenden wollen wir Pläne betrachten, die aus parallelen *Phasen* bestehen. In jeder Phase werden höchstens P Aufgaben auf jeweils gleich vielen Prozessoren berechnet. Die Phasen werden sequentiell abgearbeitet. Wenn also beispielsweise der Plan aus zwei Phasen besteht, beginnt die Bearbeitung der Aufgaben in der zweiten Phase erst dann, wenn die Aufgaben in der ersten Phase berechnet sind.

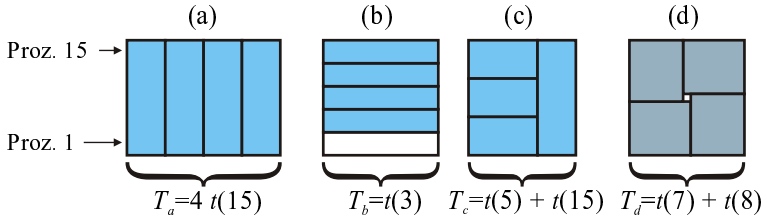


ABBILDUNG 4.3.1. Pläne für 4 Aufgaben und 15 Prozessoren. Die Pläne (a), (b) und (c) sind phasenparallele Pläne.

Abbildung 4.3.1 zeigt einige Pläne für 4 Aufgaben und 15 Prozessoren. Plan (a) besteht aus vier Phasen, Plan (b) aus einer Phase und Plan (c) aus zwei Phasen. Plan (d) ist kein phasenparalleler Plan. Allerdings könnte er optimal sein. Man kann also nicht erwarten, durch die Restriktion auf phasenparallele Pläne einen optimalen Plan zu finden.

Wegen der Phasenstruktur liegt eine andere Notation der Pläne nahe. Und zwar notieren wir einen phasenparallelen mit m Phasen für n Aufgaben durch ein m-Tupel $(n_1, \ldots, n_m)$, wobei $n_1 + \cdots + n_m = n$. Ein phasenparalleler Plan ist optimal, wenn sein Makespan unter allen phasenparallelen Plänen für dieselbe Aufgabenanzahl minimal ist. Wir können phasenparallele Pläne mit dynamischer Programmierung berechnen, indem wir die folgende Suboptimalitätseigenschaft phasenparalleler Pläne ausnutzen.

LEMMA 4.3.1. *Ein optimaler phasenparalleler Plan für n Aufgaben besteht entweder nur aus einer Phase oder er ist eine Konkatenation zweier optimaler phasenparalleler Pläne für n_1 beziehungsweise n_2 Aufgaben mit $n = n_1 + n_2$.*

(Ohne Beweis.)

THEOREM 4.3.2. *Die Zeit für die Berechnung eines optimalen phasenparallelen Plans eines GPVAs für n Aufgaben wird von n^2 dominiert.*

Algorithmus 4 Optimale phasenparallele Pläne

Eingabe: n Aufgaben, P Prozessoren, $t : \{1, \ldots, P\} \to \mathbb{R}^{>0}$
Ausgabe: ein optimaler phasenparalleler Plan $(n_1, \ldots, n_m)$

1. **for** $i = 1$ **to** n
2. **if** $i \leq P$ **then**
3. $S[i] \leftarrow \left(t\left(\left\lfloor \frac{P}{i} \right\rfloor\right), (i)\right)$
4. **else**
5. $S[i] \leftarrow (\infty, ())$
6. **for** $j = 1$ **to** $\left\lfloor \frac{i}{2} \right\rfloor$
7. **if** $S[j].t + S[i-j].t < S[i].t$ **then**
8. $S[i] \leftarrow (S[j].t + S[i-j].t,\ S[j].S \circ S[i-j].S)$
9. **return** $(S[n].S)$

BEWEIS. Der Algorithmus 4 benutzt einen Vektor $S[1 \ldots n]$ um optimale Pläne zu speichern. Jeder Eintrag $S[i]$ des Vektors besteht aus zwei Teilen, wobei $S[i].S$ ein Plan für i Aufgaben mit Makespan $S[i].t$ ist. Der Operator $\circ$ bezeichnet die Konkatenation zweier Pläne.

Die Invariante der äußeren Schleife ist die Tatsache, dass die Einträge $S[1].S, \ldots, S[i-1].S$ optimale Pläne enthalten. In der i-ten Iteration wird ein optimaler Plan für i Aufgaben berechnet. Hierbei wird die Aussage von Lemma 4.3.1 ausgenutzt. Durch die Fallunterscheidung in Zeile 2 werden Pläne mit einer Phase berücksichtigt. Diese gibt es jedoch nur, wenn höchstens so viele Aufgaben wie Prozessoren vorhanden sind. Da die Laufzeitfunktion monoton fallend ist, ergibt sich ein optimaler Plan mit nur einer Phase dann, wenn jeder Aufgabe $\lfloor P/i \rfloor$ Prozessoren zugeordnet werden. Schließlich werden in der inneren Schleife alle Zerlegungen von i in zwei Summanden j und $i-j$ betrachtet. Da beide Summanden kleiner als i sind, sind die zugehörigen optimalen phasenparallelen Pläne bereits berechnet. □

THEOREM 4.3.3. *Es sei R ein GPVA dessen Laufzeitfunktion die Bedingung $i \leq j \Rightarrow i\,t(i) \leq j\,t(j)$ erfüllt. Es sei S ein optimaler phasenparalleler Plan für R und es sei O ein optimaler Plan für R. Dann gilt $M(S) \leq 2\,M(O)$.*

BEWEIS. Es sei n die Anzahl der Aufgaben. Wir unterscheiden zwei Fälle.

FALL 1*: $n \geq P$.* Die Gesamtarbeit jedes Plans ist mindestens $n\,t(1)$ und somit ist $M(O) \geq n\,t(1)/P$. Wir können einen phasenparallelen Plan für die n Aufgaben so erstellen, dass in den ersten $\lfloor n/P \rfloor$ Phasen jeweils P Aufgaben berechnet werden und die restlichen $n \mod P$ Aufgaben in einer weiteren Phase. Da S ein optimaler phasenparalleler Plan ist, gilt

$$M(S) \leq \left\lfloor \frac{n}{P} \right\rfloor\, t(1) + t\left(\left\lfloor \frac{P}{n \mod P} \right\rfloor\right) \leq \left(\frac{n}{P} + 1\right)\, t(1) \ .$$

Es ist somit

$$\frac{M(S)}{M(O)} \leq 1 + \frac{P}{n} \leq 2 \ .$$

FALL 2: $n < P$. Es gibt einen phasenparallelen Plan mit einer Phase dessen Makespan durch $t(\lfloor P/n \rfloor)$ gegeben ist. Da ein optimaler Plan höchstens diesen Makespan hat, arbeiten in einem optimalen Plan mindestens $\lfloor P/n \rfloor$ Prozessoren an jeder Aufgabe. Jede Aufgabe verursacht also mindestens die Arbeit $\lfloor P/n \rfloor\ t(\lfloor P/n \rfloor)$. Die Gesamtarbeit ist somit mindestens $n\ \lfloor P/n \rfloor\ t(\lfloor P/n \rfloor)$ und folglich ist

$$M(O) \geq \frac{n}{P}\ \lfloor P/n \rfloor\ t(\lfloor P/n \rfloor) \ .$$

Es folgt wegen $M(S) \leq t(\lfloor P/n \rfloor)$ für den Approximationsfaktor

$$\frac{M(S)}{M(O)} \leq \frac{\frac{P}{n}}{\lfloor \frac{P}{n} \rfloor} \leq 2 \ .$$

□

4.4. Baumstrukturierte Berechnungen mit parallelisierbaren Knoten

Die parallele Berechnung baumstrukturierter Anwendungen kann beispielsweise dadurch geschehen, dass Knoten, die sich in derselben Tiefe des Baumes befinden, gleichzeitig berechnet werden. Wenn jedoch die Bäume schmal sind, reicht die Nutzung dieser *Baumparallelität* nicht aus, um eine gute Skalierbarkeit zu erreichen. Eine Möglichkeit, auch bei schmalen Bäumen einen skalierbaren Algorithmus zu erhalten ist die, die Berechnung jedes einzelnen Baumknotens zu parallelisieren. Indem die Baumebenen nacheinander bearbeitet werden, entsteht für jede Ebene ein PVA oder, falls die Knotenberechnungen gleiche Bearbeitungszeiten haben, ein GPVA.

Im Folgenden wird die Isoeffizienz von Baumberechnungen betrachtet, deren Knoten dieselbe Bearbeitungszeit und Skalierbarkeit haben. Für die dabei entstehenden GPVA Probleme werden phasenparallele Pläne benutzt. Die Konkatenation der Pläne ist wieder ein phasenparalleler Plan, so dass die komplette Baumberechnung eine *phasenparallele Berechnung* ist.

DEFINITION 4.4.1. Es sei S ein phasenparalleler Plan. Die *Breite* von S ist die maximale Anzahl von Aufgaben, die in derselben Phase bearbeitet werden. Die Anzahl der Aufgaben wird durch $|S|_1$ bezeichnet.

LEMMA 4.4.2. *Wenn eine phasenparallele Berechnung die Effizienz e hat, dann enthält sie eine Phase, die mindestens die Effizienz e hat. Wenn eine Phase die Effizienz e' hat, dann werden alle in ihr enthaltenen Aufgaben mindestens mit Effizienz e' berechnet.*

BEWEIS. Eine phasenparallele Berechnung $(n_1, \ldots, n_m)$ habe die Effizienz e. Es sei t_i die Berechnungszeit der Phase i und es sei e_i ihre Effizienz. Es sei j eine Phase mit maximaler Effizienz. Dann gilt

$$e = \frac{e_1 t_1 + \cdots + e_m t_m}{t_1 + \cdots + t_m} \leq \frac{e_j\,(t_1 + \cdots + t_m)}{t_1 + \cdots + t_m} = e_j \quad .$$

Um die zweite Aussage zu zeigen, betrachten wir eine Phase mit n Aufgaben und Effizienz e'. Es sei P die Anzahl der Prozessoren und es sei t(p) die Berechnungszeit eines Knotens auf p Prozessoren. Es sei e'' die Effizienz einer Aufgabe. Dann gilt $e' = n\,t(1)/\left(P\,t(\lfloor n/P \rfloor)\right) \leq t(1)/\left(\lfloor P/n \rfloor\ t(\lfloor n/P \rfloor)\right) = e''$. □

DEFINITION 4.4.3. Es sei x eine Eingabe und es sei P die Anzahl der Prozessoren. Dann bezeichnet $\mathcal{S}(x,\ P)$ den phasenparallelen Plan, der benutzt wird, um x auf P Prozessoren zu berechnen. Die Berechnungszeit, die Effizienz und die Isoeffizienz wird durch $T^{(\mathcal{S})}\,(x,\ P)$, $E^{(\mathcal{S})}(x,\ P)$ beziehungsweise durch $I_e^{(\mathcal{S})}(P)$ bezeichnet.

THEOREM 4.4.4. *Es sei X eine Eingabemenge auf der alle phasenparallelen Pläne höchstens die Breite N haben. Es sei $I_e(P)$ die Isoeffizienzfunktion der Aufgaben. Wenn die Berechnungszeiten der Aufgaben diskret sind und die Effizienzfunktion E der Aufgaben die speed-up und die size-up Bedingungen erfüllt, dann gilt für alle $P \geq N$*

$$I_e^{(\mathcal{S})}(P) \geq I_e(\lfloor P/N \rfloor).$$

BEWEIS. Da die Berechnungszeiten diskret sind, ist auch die Baumberechnungszeit $T^{(\mathcal{S})}$ diskret. Wegen Theorem 2.2.5 gibt es eine Eingabe x so dass $I_e^{(\mathcal{S})}(P) = T^{(\mathcal{S})}(x,\ 1)$ und $E^{(\mathcal{S})}(x, 1) \geq e$. Wenn $T(x,\ 1)$ die sequentielle Berechnungszeit einer Aufgabe ist, gilt $T^{(\mathcal{S})}(x,\ 1) \geq T(x,\ 1)$, da der Plan aus mindestens einer Aufgabe besteht. Die effizienteste Aufgabenberechnung benutzt $P' \geq \lfloor P/N \rfloor$ Prozessoren und hat wegen Lemma 4.4.2 eine Effizienz $e' \geq e$ (also $E(x,\ P') = e'$). Da E die size-up Eigenschaft erfüllt, gilt $T(x,\ 1) = I_{e'}(P')$ (Theorem 2.2.9(1)). Da die Isoeffizienz monoton in e ist, gilt $I_{e'}(P') \geq I_e(P')$. Wegen der speed-up Eigenschaft von E und Theorem 2.2.7 gilt $I_e(P') \geq I_e(\lfloor P/N \rfloor)$. Insgesamt,

$$I_e^{(\mathcal{S})}(P) = T^{(\mathcal{S})}(x,\ 1) \geq T(x,\ 1) \geq I_e(\lfloor P/N \rfloor) \quad .$$

□

THEOREM 4.4.5. *Es sei X eine Eingabemenge auf der alle phasenparallelen Pläne höchstens aus N Aufgaben bestehen. Es sei $I_e(P)$ die Isoeffizienzfunktion der Aufgaben. Wenn die*

Berechnungszeiten diskret sind und die Effizienzfunktion der Aufgaben E die speed-up und size-up Eigenschaften hat, dann gilt

$$I_e^{(\mathcal{S})}(P) \leq N\, I_e(P) \ .$$

BEWEIS. Wegen der Diskretheit und Theorem 2.2.5 gibt es eine Eingabe x mit $I_e^{(\mathcal{S})}(P) = T^{(\mathcal{S})}(x,\, 1)$. Die Aufgabe mit der kleinsten Effizienz benutzt $P' \leq P$ Prozessoren und hat höchstens die Effizienz e, d.h. $E(x,\, P') \leq e$. Da E die size-up Eigenschaft hat, erhalten wir mit Theorem 2.2.1, dass $T(x,\, 1) \leq I_e(P')$. Durch die speed-up Eigenschaft von E und wegen Theorem 2.2.7 gilt $I_e(P') \leq I_e(P)$. Schließlich,

$$I_e^{(\mathcal{S})}(P) = T^{(\mathcal{S})}(x,\, 1) \leq N\, T(x,\, 1) \leq N\, I_e(P') \leq N\, I_e(P) \ .$$

□

4.5. Offene Fragen

Bezüglich des GPVA Problems bleiben zwei Fragen offen.

Erstens, wie ist die Komplexität, einen optimalen Plan für das GPVA zu bestimmen, einzuordnen? Und zweitens, wie gut sind phasenparallele Pläne?

In Theorem 4.3.3 haben wir gezeigt, dass der Makespan phasenparalleler Pläne höchstens doppelt so lang ist wie der eines optimalen Plans. Dieser Faktor wird aber bereits von sehr einfachen Plänen erreicht, die in linearer Zeit angegeben werden können. Es ist daher zu vermuten, dass der Approximationsfaktor von phasenparallelen Plänen besser als zwei ist. Betrachten wir hierzu noch einmal die in Abbildung 4.3.1 dargestellte Situation. Falls wir von einer monotonen Zunahme der Arbeit bei wachsender Prozessoranzahl ausgehen, erhalten wir $t(8) \geq 7/8\, t(7)$ und somit $T_d \geq (7/8 + 1)\, t(7)$. Weiterhin erhalten wir $t(5) \leq 7/5\, t(7)$, und wegen $t(15) \leq t(7)$ die Ungleichung $T_c \leq (7/5 + 1)\, t(t)$. Falls also (c) ein optimaler phasenparalleler Plan und (d) ein optimaler Plan ist, ist der Approximationsfaktor T_c/T_d höchstens $32/25$. Falls (b) ein optimaler phasenparalleler Plan ist, erhalten wir mit $t(3) \leq 7/3\, t(7)$ eine obere Schranke von $56/45$.

In Bezug auf Abschnitt 4.4 ist es interessant, weitere allgemeine Aussagen über die Isoeffizienz phasenparalleler Anwendungen herzuleiten. Was ist zum Beispiel die Isoeffizienz von Anwendungen, deren Aufgabengraphen vollständige Binärbäume sind? Eng mit dieser Frage verknüpft ist die Frage, ob es eine Polynomklasse gibt, bei der die Größenordnung der Isoeffizienz der Descartes Methode besser als quadratisch ist.

KAPITEL 5

Statische Lastbalancierung unabhängiger Lastelemente

Die im letzten Kapitel behandelten Lastverteilungstechniken finden vor allem dann Verwendung, wenn weniger parallele Aufgaben als Prozessoren vorhanden sind. Dieses Kapitel befasst sich mit der Situationen, in der mehr Aufgaben als Prozessoren vorhanden sind. Wir werden davon ausgehen, dass die Aufgaben initial auf den Prozessoren verteilt sind und dass jede Aufgabe dieselbe Bearbeitungszeit hat. Die Aufgabe der Lastbalancierung ist es, die Aufgaben so umzuverteilen, dass am Ende jedem Prozessor gleich viele Aufgaben zugeordnet sind. Hierbei werden während der Balancierungsphase weder Aufgaben bearbeitet noch generiert.

Sowohl das verwandte Problem der *Tokenverteilung*, bei dem pro Schritt nur ein Lastelement transportiert werden kann, als auch das der Lastbalancierung selbst ist intensiv untersucht worden. Wir werden uns in diesem Abschnitt auf eine bestimmte Klasse von Lastbalancierungsalgorithmen, den Diffusionsverfahren, konzentrieren. Es handelt sich bei diesen Verfahren um *lokale Iterationsverfahren*; also um Verfahren, die in mehreren Iterationen ablaufen und bei denen in jeder Iteration jeder Prozessor nur jeweils mit einer begrenzten Menge von Nachbarprozessoren Nachrichten austauscht. Diese Nachbarschaftsstruktur ist fest und kann durch einen ungerichteten Graphen, der *Balancierungstopologie* definiert werden—nur Prozessoren, die in dieser Topologie benachbart sind, tauschen während der Lastbalancierung Botschaften aus.

Es sind im wesentlichen die folgenden drei Gesichtspunkte, die die Wahl der Topologie beeinflussen.

(1) *Eignung für die Anwendung.* In einigen Anwendungen gibt es starke Abhängigkeiten zwischen den Aufgaben. Ein Beispiel sind adaptive numerische Simulationen. In diesem Fall sollte die Topologie so gewählt werden, dass sie die Abhängigkeiten berücksichtigt, und somit eine lokalitätserhaltende Lastbalancierung ermöglicht.
(2) *Eignung für die Rechnerarchitektur*. Die Topologie kann so gewählt werden, dass sie die Struktur des Kommunikationsnetzwerks wiederspiegelt. Dieses führt zu physikalisch lokaler Kommunikation und reduziert somit die gesamte von der Lastbalancierung erzeugten Kommunikationslast.
(3) *Eignung für die Lastbalancierung.* Die Konvergenz der hier betrachteten Lastbalancierungsverfahren hängt entscheidend von der gewählten Topologie ab. Insbesondere können manche Verfahren nur bei bestimmten Topologien sinnvoll eingesetzt werden. Die Topologie kann also auch so gewählt werden, dass die Lastbalancierung schnell abgewickelt werden kann.

Da in Hinblick auf das Lastverteilungssystem VDS Anwendungen, die unabhängige Aufgaben generieren, von besonderem Interesse sind, und der zweite Aspekt bei massiv parallelen Rechnern oder bei modernen Clustersystemen nur eine untergeordnete Rolle spielt, wird in diesem Kapitel der dritte Aspekt im Mittelpunkt stehen.

Die Verwendung lokaler Iterationsverfahren legt nahe, in jedem Iterationsschritt Lastverschiebungen zwischen benachbarten Prozessoren vorzunehmen. Bei diesem Vorgehen wird jedoch typischerweise deutlich mehr Last migriert als eigentlich notwendig ist [**44**]. Aus diesem Grund wird der Lastbalancierungsprozess in zwei Phasen aufgeteilt. Die erste Phase berechnet die Anzahl der Aufgaben, die jeweils zwischen zwei Prozessoren verschickt werden müssen und arbeitet mit einem virtuellen Lastmaß, dass beliebige (auch negative) Werte annehmen darf. Da während dieser Phase ein Lastfluss berechnet wird, nennen wir sie *Flussberechnungsphase*. Erst in der zweiten Phase, der *Migrationsphase*, werden die Aufgaben gemäß des berechneten Lastflusses verschoben.

Algorithmen für die Lastflussberechnung wurden in den letzten Jahren intensiv untersucht. Viele von ihnen sind lokale Iterationsverfahren. Sie unterscheiden sich unter anderem darin, ob sie die Nachbarn eines Prozessors in einer Iteration teilweise oder vollständig betrachten. Ein Beispiel für die teilweise Betrachtung der Nachbarn ist das *Dimension Exchange* Verfahren, bei dem jeweils nur ein Nachbar betrachtet wird [**36, 138**]. Auf *Diffusion* basierende Verfahren betrachten dagegen alle Nachbarn [**36, 44, 50, 64, 71, 70, 138**]. Ein Vorteil dieser Verfahren ist, dass sie alle den eindeutig bestimmten l_2-minimale Fluss bestimmen [**44**]. Dieses ist bei der teilweisen Betrachtung der Nachbarn im Allgemeinen nicht der Fall.

In diesem Kapitel wird neben dem *optimalen* Diffusionsverfahren *OPT* [**50**] das *iterierte* Diffusionsverfahren *OPT-IT* betrachtet. OPT-IT kann ausschließlich bei Topologien angewendet werden, die sich als kartesische Produkte anderer Topologien darstellen lassen. Hierbei ist die Idee von OPT-IT, die Faktortopologien nacheinander mit dem OPT Verfahren zu balancieren. Durch die getrennte Betrachtung der Faktortopologien wird jeweils nur ein Teil der benachbarten Prozessoren in jeder Iteration betrachtet, wodurch der resultierende Lastfluss nicht notwendigerweise l_2-minimal ist. Der Vorteil von OPT-IT liegt allerdings darin, dass weniger Iterationen als bei der Verwendung von OPT für die vollständige Topologie benötigt werden. Es stellt sich also neben der anfangs aufgeworfenen Frage nach einer geeigneten Topologie auch die nach der Wichtigkeit der l_2-Minimalität des Diffusionsflusses.

In den bisherigen Arbeiten wurden die Verfahren oft mit Hilfe von Simulationen, wie sie beispielsweise durch die PARTY-Bibliothek [**110**] zur Verfügung gestellt werden, bewertet [**44, 50**]. Diese Untersuchungen eignen sich gut für die Beurteilung der Konvergenzeigenschaften, können jedoch den tatsächlichen Kommunikationsaufwand nur annähernd modellieren. Mit dem Ziel, diesen Aufwand realistisch zu beurteilen, führen wir hier Experimente mit einer verteilten Implementierung der Verfahren mit Hilfe von VDS durch.

Um sowohl die Charakteristik von eng gekoppelten als auch von lose gekoppelten Systemen berücksichtigen zu können, wurden als Experimentierplattformen eine Cray T3E und ein Workstation-Cluster (HPCLine) gewählt. Die T3E verkörpert die typischen Eigenschaften eines massiv parallelen Systems, da ihre MPI Implementierung sehr kurze Startup-Zeiten und geringe Latenzzeiten erzeugt. Das verwendete Clustersystem besteht aus 96 Doppelprozessor-Knoten (Pentium II, 450 MHz), die durch Ethernet verbunden sind. In den Experimenten wird jeweils pro Knoten ein Rechenprozess gestartet. Die Kommunikation basiert in diesem Fall auf PVM. Die Latenzzeit des HPCLine-Systems ist etwa um Faktor 15 länger als die der Cray.

In Abschnitt 5.1 werden zunächst grundlegende Definitionen eingeführt. Auf den Stand der Forschung wird in Abschnitt 5.2 eingegangen. In Abschnitt 5.3 wird die Flussberechnung genauer untersucht. Hierbei stellt sich heraus, dass der Aufwand im wesentlichen vom maximalen Knotengrad und von der Anzahl der Eigenwerte der zur Topologie gehörigen Laplacematrix abhängt.

Der Abschnitt 5.4 befasst sich mit der Migrationsphase. Hier betrachten wir die *Proportionalheuristik*, die Aufgaben vorzugsweise an Nachbarn schickt, die am meisten Last zu bekommen haben [**44**]. Wir zeigen, dass sowohl ein kleiner Gesamtfluss als auch ein kleiner Durchmesser wichtige Parameter für die Dauer dieser Phase sind. In Abschnitt 5.5 betrachten wir die gesamten Balancierungskosten und zeigen einen Tradeoff zwischen dem Aufwand der in die Flussberechnung investiert wird (und somit der Flussqualität) und der Dauer der Balancierungsdauer auf. Das Kapitel schließt mit einem Vergleich des Diffusionsverfahrens und des Workstealing-Verfahrens im Rahmen einer konkreten Anwendung aus dem OR-Bereich.

5.1. Definitionen

DEFINITION 5.1.1. Wir stellen die *Topologie* als ungerichteten Graphen $G = (V, E)$ mit $|V| = n$ Knoten und $|E| = N$ Kanten dar. Mit $\deg(G)$ bezeichnen wir den maximalen Knotengrad und mit D den Durchmesser von G. Der gerichtete Graph $\vec{G} = (V, \vec{E})$ ensteht aus G, indem jede Kante mit einer beliebigen aber festen Richtung versehen wird. Wir bezeichnen mit $Z \in \{-1, 0, 1\}^{n \times N}$ die *Knoten-Kanten Inzidenzmatrix* von $\vec{G}$. Jede Zeile dieser Matrix enthält genau einen positiven und einen negativen Eintrag. Und zwar ist $Z[e,\, i] = 1$ und $Z[e,\, j] = -1$ genau dann wenn $e = (i,\, j) \in \vec{E}$.

DEFINITION 5.1.2. Wir bezeichnen mit $w_i \in \mathbb{R}^{\geq 0}$ die *initiale Last* eines Knotens $v_i \in V$ und mit $w := (w_1, \ldots, w_n)$ den *initialen Lastvektor*. Mit $\tilde{w} := \frac{1}{n} \sum_{i=1}^{n} w_i$ bezeichnen wir die *Durchschnittslast* und mit $\overline{w} := (\tilde{w}, \ldots, \tilde{w})$ den *ausgeglichenen Lastvektor*.

DEFINITION 5.1.3. Wir bezeichnen durch $x_e \in \mathbb{R}$ den *Lastfluss einer Kante* $e \in \vec{E}$ und mit $x \in \mathbb{R}^N$ den *Vektor der Kantenflüsse*. Wir nennen x einen *balancierenden Fluss* für die initiale Last w, wenn für alle $1 \leq i \leq n$

$$w_i + \sum_{e=(v_j,v_i)\in\vec{E}} x_e - \sum_{e=(v_i,v_j)\in\vec{E}} x_e = \tilde{w}$$

gilt.

Für einen balancierenden Fluss x für w gilt somit

$$Z\,x = w - \overline{w}\ . \tag{5.1.1}$$

Wir betrachten die Qualitätsmaße

- $l_1(x) = \|x\|_1 = \sum_{e\in\vec{E}} |x_e|$ (Gesamtkommunikationskosten),
- $l_2(x) = \|x\|_2 = \sqrt{\sum_{e\in\vec{E}} x_e^2}$ (Gleichmäßigkeit der Kantenauslastung),
- $l_\infty(x) = \|x\|_\infty = \max_{e\in\vec{E}} |x_e|$ (das maximale Kommunikationsvolumen zwischen zwei Knoten),
- den *Knotenfluss* $f(x) = \max_{v_i\in V} f(v_i)$ mit $f(v_i) = \sum_{e=\{v_i,v_j\}\in E} |x_e|$ (die Summe aus ein- und ausgehendem Fluss, vgl. [**116**]).

Um eine geeignete Darstellung der Situation während der Migrationsphase zu erhalten, führen wir einen weiteren gerichteten Graphen mit der Knotenmenge V ein, dessen Kanten in die Richtung des Lastflusses zeigen.

DEFINITION 5.1.4. Der gerichtete *Migrationsgraph* $M_G(x) = (V, \tilde{E})$ ist ein Graph, der die Kante (v_i, v_j) genau dann enthält, wenn $(v_i, v_j) \in \vec{E} \wedge x_{(v_i,v_j)} > 0$ oder $(v_j, v_i) \in \vec{E} \wedge x_{(v_j,v_i)} <$

0. Der *Migrationsfluss* $\tilde{x} \in \mathbb{R}^N$ auf $\tilde{E}$ wird definiert durch $\tilde{x}_{(v_i,v_j)} := x_{(v_i,v_j)}$ falls $(v_i, v_j) \in \vec{E}$, und $\tilde{x}_{(v_i,v_j)} = -x_{(v_j,v_i)}$ sonst.

Es ist offensichtlich, dass im Falle von l_1-minimalen und l_2-minimalen Flüssen der Migrationsgraph kreisfrei ist.

DEFINITION 5.1.5. Es sei $A \in \{0,1\}^{n\times n}$ die symmetrische Adjazenzmatrix von G. Die *Laplacematrix* $L \in \mathbb{Z}^{n\times n}$ von G ist definiert durch

$$L := \text{diag}\,(\deg(v_1), \ldots, \deg(v_n)) - A \ .$$

Wir bezeichnen mit m die Anzahl der unterschiedlichen Eigenwerte von L. Wir werden diese Eigenwerte $0 = \lambda_1 < \cdots < \lambda_m$ im Folgenden auch als *Eigenwerte des Graphen* G bezeichnen.

DEFINITION 5.1.6. Es seien G_1 und G_2 ungerichtete Graphen. Das *kartesische Produkt* $G = G_1 \times G_2$ hat die Knotenmenge $V(G) = V(G_1) \times V(G_2)$ und die Kantenmenge

$$E = \{\{(u_i, u_j), (v_i, v_j)\} \mid u_i = v_i \wedge \{u_j, v_j\} \in E(G_2) \text{ oder } u_j = v_j \wedge \{u_i, v_i\} \in E(G_1)\} \ .$$

5.2. Stand der Forschung

Eine Möglichkeit, den Fluss zu bestimmen, resultiert direkt aus der Eigenschaft (5.1.1) eines balancierenden Flusses. Hu, Blake und Emerson lösen zunächst das Gleichungsssystem $Ly = w - \overline{w}$ direkt und erhalten somit den Fluss, indem sie $x = Z^T y$ setzen [**71**]. Diese Berechnung zentral auf einem Knoten auszuführen verlangt das Versenden großer Datenmengen beim Sammeln der Lastinformationen und beim Verbreiten des berechneten Flusses. Ein weiterer Ansatz, der in dieser Arbeit vorgeschlagen wird, ist das System $Zx = w - \overline{w}$ verteilt mittels eines CG-Verfahrens zu lösen. Allerdings benötigt auch dieser Ansatz drei globale Summierungen skalarer Werte.

Die andere Möglichkeit geht eher auf die eigentliche Definition des balancierenden Flusses zurück. Hier wird der Fluss durch Anwendung eines (iterativen) Lastbalancierungsverfahrens wie Diffusion oder Dimension Exchange ermittelt. Auf die Diffusionsverfahren wird im folgenden Abschnitt noch genauer eingegangen. Sie haben den Vorteil, dass sie den eindeutigen l_2-minimalen Fluss berechnen [**44**]. Der von ihnen erzeugte Migrationsgraph ist somit kreisfrei. Der von Dimension Exchange berechnete Fluss ist dagegen nicht notwendigerweise l_2-minimal und hängt von der Reihenfolge, in der die Nachbarn eines Prozessors betrachtet werden, ab.

Das *Alternating Direction Iterative*-Verfahren (*ADI*) wendet eine Methode zum Lösen linearer Gleichungssysteme [**130**] auf das Lastbalancierungsproblem an [**50**]. Es stellt eine Kombination aus Diffusion und Dimension Exchange dar und kann immer dann angewendet werden, wenn sich die Topologie als kartesisches Produkt darstellen lässt. Falls $G = G_1 \times G_2$ ist, dann führt jeder Knoten in jeder Iteration zunächst mit seinen Nachbarn bezüglich G_1 Balancierungsschritte aus und dann mit denen aus G_2. Der resultierende Fluss kann allerdings Zyklen enthalten, was zu einem sehr großen Gesamtfluss führt.

5.2.1. Diffusionsverfahren. In dem von Cybenko eingeführten FOS-Verfahren [**36**] ergibt sich die Last w_i^k von Knoten $v_i \in V$ in Iteration k aus

$$w_i^k = w_i^{k-1} - \sum_{\{v_i,v_j\}\in E} \alpha \left(w_i^{k-1} - w_j^{k-1}\right) \ , \tag{5.2.1}$$

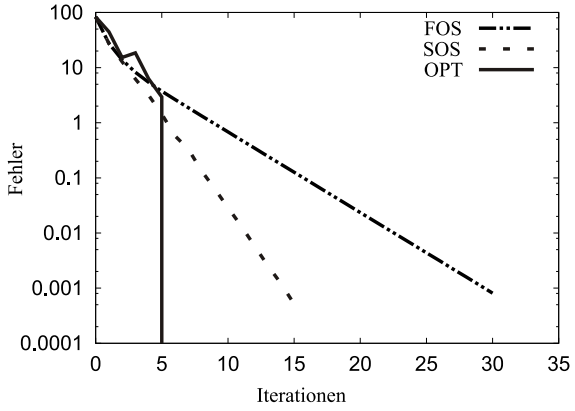


ABBILDUNG 5.2.1. Simulativer vergleich der drei Diffusionsverfahren FOS, SOS und OPT. Die Topologie ist in diesem Fall eine Hypercube der Dimension 6. Die Ausgangslast wurde durch die sukzessive Zuordnung von 6400 Lastelementen an zufällig ausgewählte Prozessoren generiert. Der Fehler berechnet sich aus $||w^k - \overline{w}||$.

und der akkumulierte Fluss bezüglich einer Kante $e = \{v_i, v_j\}$ aus

$$x_e^k = x_e^{k-1} + \alpha \left(w_i^{k-1} - w_j^{k-1} \right) \quad .$$

Für eine geeignete Wahl des Diffusionsparameters α, konvergiert das FOS Verfahren gegen die Durchschnittslast $\overline{w}$. Indem wir die Diffusionsmatrix $M := I - \alpha L \in \mathbb{R}^{n \times n}$ definieren, können wir Gleichung (5.2.1) schreiben als

$$w^k = M w^{k-1} \quad .$$

Das FOS Verfahren konvergiert am schnellsten, indem $\alpha = 2/(\lambda_2 + \lambda_m)$ gewählt wird. Es hat allerdings den Nachteil, relativ langsam zu konvergieren. Gosh u.a. benutzen das Konzept der Überrelaxierung um die Konvergenz zu beschleunigen [**64**]. In dem *Second-Order-Scheme* (*SOS*) hängt die Last, die zwischen zwei Prozessoren ausgetauscht wird, nicht nur von der aktuellen Lastdifferenz, sondern auch von der Differenz in der vorangegangenen Iteration ab. Eine weitere Verbesserung der Konvergenz wird durch das polynomielle Tschebyscheff-Schema erreicht [**43, 70**]. Jedoch ist die Konvergenz von diesem Schema asymtotisch mit der von SOS gleichzusetzen [**43**]. Mit beiden Ansätzen kann zwar die Konvergenz von FOS verbessert werden, jedoch hängt die Dauer der Flussberechnung jeweils von der initialen Lastsituation ab.

Eine deutliche Verbesserung stellen die *optimalen* Diffusionsverfahren dar, die unabhängig von der initialen Lastsituation, nur $m - 1$ Iterationen benötigen. Da es Lastsituationen gibt, bei denen mindestens D Schritte erforderlich sind, und die optimalen Verfahren stets $m - 1$ Schritte benötigen, ist eine unmittelbare Konsequenz aus der Existenz dieser Verfahren die Tatsache, dass $D \leq m - 1$.

Diekmann, Frommer und Monien stellen in [**43, 44**] das erste numerisch stabile optimale Verfahren vor (*OPS*). Das von Elsässer u.a. vorgestellte optimale Verfahren *OPT* ist eine Vereinfachung von OPS [**50**]. Es ist dem FOS Verfahren sehr ähnlich, wählt jedoch in jeder Iteration einen anderen Diffusionsparameter. In Iteration k wird $\alpha_k = 1/\lambda_{\pi(k+1)}$ gesetzt, wobei π eine

Permutation auf dem Menge $\{2, \ldots, m\}$ ist, die die numerische Stabilität des Verfahrens gewährleistet. Der Vergleich in Abbildung 5.2.1 zeigt, dass OPT wesentlich schneller konvergiert als FOS und SOS. Für das OPT Verfahren werden zwar im Gegensatz zu FOS und SOS alle Eigenwerte der Laplacematrix benötigt, jedoch kann dieses im Falle einer festen, von der Anwendung unabhängigen Topologie in Kauf genommen werden.

Im nächsten Abschnitt diskutieren wir die Kriterien, die bei der Auswahl einer geeigneten Topologie in Bezug auf die Flussberechnung beachtet werden sollten.

5.3. Flussberechnung

Da das OPT Verfahren $m-1$ Iterationen durchführt und in jeder Iteration jeder Knoten jeweils mit allen Nachbarn Nachrichten austauscht, gilt für die Laufzeit T_{OPT} des Verfahrens

$$T_{\text{OPT}} \leq 2\,(m-1)\,\deg(G)\,o + (m-1)\,L \quad . \tag{5.3.1}$$

Algorithmus 5 Das iterierte OPT Verfahren (OPT-IT).

Eingabe: Eine Topologie $G = (V,\, E) = G_1 \times \cdots \times G_d$,
ein Knoten $v \in G$, die initiale Last l von v,
für $1 \leq i \leq d$: die Eigenwerte $\lambda_2^{(i)}, \ldots, \lambda_{m_i}^{(i)}$ von G_i,
eine Permutation $\pi^{(i)}$ auf der Menge $\{2, \ldots, m_i\}$.

Ausgabe: Die Werte $x_{(v,\,w)}$ für $\{v,\, w\} \in E$ eines balancierenden Flusses x.

1. $x_{(v,\,w)} \leftarrow 0$ für alle $\{v,\, w\} \in E$
2. for $i = 1$ to d
3. for $j = 2$ to m_i
4. sende l zu allen Nachbarn $w \in \{w_1, \ldots, w_k\}$ von v in G_i
5. for all $w \in \{w_1, \ldots, w_k\}$
6. empfange die Lastinformation l_w von Nachbar w
7. $\delta \leftarrow (1/\lambda_{\pi^{(i)}(j)})\,(l - l_w)$
8. $l \leftarrow l - \delta$
9. $x_{(v,\,w)} \leftarrow x_{(v,\,w)} + \delta$

Wir können diese Laufzeit reduzieren, indem wir Topologien benutzen, die sich als (nicht triviales) kartesisches Produkt darstellen lassen. In diesem Fall kann ein balancierender Fluss berechnet werden, indem die Faktortopologien getrennt voneinander betrachtet werden. Für die Berechnung der Flüsse innerhalb der Faktortopologien kann beispielsweise das OPT-Verfahren verwendet werden. Wir sprechen in diesem Fall von dem iterierten OPT Verfahren (OPT-IT, Algorithmus 5).

Es sei beispielsweise $G = G_1 \times G_2 = (V,\, E)$. Dann ist jeder Knoten $(v,\, w) \in G$ Teil eines Graphen $G_{(\bullet,w)}$, der isomorph zu G_1 ist. Hierbei besteht die Knotenmenge von $G_{(\bullet,w)}$ gerade aus den Knoten $\{(v,\, w) \mid (v,\, w) \in V\}$. Wir nennen die Graphen $G_{(\bullet,w)}$ für $w \in V(G_2)$ *Graphen der ersten Dimension.* Analog ist der Knoten $(v,\, w)$ auch Teil eines zu G_2 isomorphen *Graphen der zweiten Dimension* $G_{(v,\bullet)}$.

In OPT-IT kommuniziert ein Knoten $(v,\, w)$ zunächst ausschließlich mit den Nachbarn aus $G_{(\bullet,w)}$, und zwar solange, bis die Last in $G_{(\bullet,w)}$ balanciert ist. Anschließend kommuniziert er mit den Nachbarn aus $G_{(v,\bullet)}$. In der ersten Dimension ergeben sich somit $|V(G_2)|$ und in der zweiten Dimension $|V(G_1)|$ unabhängige Lastbalancierungsprobleme. Die Probleme in der zweiten

Dimension sind zudem identisch. Falls also $G = G_1 \times \cdots \times G_d$, und die Anzahl der Eigenwerte von G_i für $1 \leq i \leq d$ durch m_i gegeben ist, erhalten wir für die Laufzeit von OPT-IT

$$T_{\text{OPT-IT}} \leq \sum_{i=1}^{d} 2\,(m_i - 1)\,\deg(G_i)\,o + (m_i - 1)\,L \ . \tag{5.3.2}$$

Der Fluss von OPT-IT ist im Gegensatz zu OPT im Allgemeinen nicht l_2-minimal. Er ist allerdings kreisfrei.

THEOREM 5.3.1. *Es sei G ein kartesisches Produkt $G = G_1 \times \cdots \times G_d$ und es sei x ein balancierender Fluss für G, der mit dem OPT-IT Verfahren berechnet worden ist. Dann ist der Migrationsgraph $M_G(x)$ kreisfrei.*

BEWEIS. Wir beweisen den Satz zunächst für $d = 2$. Es sei $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ und $G = (V, E) = G_1 \times G_2$. Es sei $w \in V_2$ beliebig aber fest. Wir betrachten den Teilgraph $M_{G_{(\bullet,w)}}(x)$ von $M_G(x)$. Da der Fluss in diesem Teilgraph durch OPT berechnet wird, ist er l_2-optimal und somit ist $M_{G_{(\bullet,w)}}(x)$ kreisfrei. Dasselbe gilt für die Teilgraphen $\{M_{G_{(v,\bullet)}}(x) \mid v \in V_1\}$ in der zweiten Dimension.

Nach dem ersten Durchlauf der äußeren Schleife von OPT-IT sind alle Teilgraphen der ersten Dimension balanciert. Die Verteilung der Last in den Teilgraphen der zweiten Dimension ist somit identisch. Da in jedem dieser Teilgraphen folglich derselbe Fluss berechnet wird, sind die Graphen $\{M_{G_{(v,\bullet)}}(x) \mid v \in V_1\}$ isomorph.

Angenommen, G habe einen Kreis. Dann muss mindestens eine Kante e des Kreises in einem Teilgraph der zweiten Dimension liegen, da die Teilgraphen der ersten Dimension kreisfrei sind. Es sei $e = ((v, w), (v, w')) \in E(M_{G_{(v,\bullet)}}(x))$. Da auch die Teilgraphen der zweiten Dimension kreisfrei sind, muss der Kreis eine Kante $((v', w'), (v', w)) \in E(M_{G_{(v',\bullet)}}(x))$ für ein $v' \in V_1$ enthalten. Wegen der Isomorphie der Graphen $\{M_{G_{(v,\bullet)}}(x) \mid v \in V_1\}$ muss $M_{G_{(v',\bullet)}}(x)$ aber auch die Kante $((v', w), (v', w'))$ enthalten. Dieses ist ein Widerspruch zur Kreisfreiheit der Graphen $\{M_{G_{(v,\bullet)}}(x) \mid v \in V_1\}$.

Wenn d größer als 2 ist, so zeigt man, dass ein Kreis Kanten aus allen Dimensionen enthalten muss und führt diese Aussage zum Widerspruch. □

An den Laufzeitschranken für T_{OPT} und $T_{\text{OPT-IT}}$ wird deutlich, dass die Topologien einen kleinen maximalen Grad haben sollten, und dass die zugehörigen Laplacematrizzen möglichst wenig Eigenwerte haben sollten. Im Folgenden werden wir Topologien betrachten, die diese Eigenschaften besitzen. Wie in Abschnitt 5.2.1 bereits erwähnt wurde, gilt $m - 1 \geq D$. Tabelle 5.3.1 zeigt einige Graphen mit $m - 1 = D$.

Pfade, Kreise, Star-Graphen, vollständige k-partite Graphen und Cliquen haben entweder einen großen maximalen Grad oder viele Eigenwerte. Hypercubes und Lattice Graphen (Hypercube Netzwerke über einem Alphabet der Größe k) haben logarithmische Werte für beide Größen. Der Nachteil dieser beiden Netzwerke ist jedoch, dass sie nur in bestimmten Größen existieren. Die folgenden Klassen zeichnen sich durch einen kleinen Durchmesser aus.

Ein (d, D)-Graph ist ein Graph, dessen Knotenanzahl unter allen regulären Graphen mit Knotengrad d und Durchmesser D maximal ist [**31**]. Ein solcher Graph hat höchstens $\frac{d(d-1)^D-2}{d-2}$ Knoten (*Moore-Schranke*). Diese Schranke wird bei Cliquen angenommen. Für $d \geq 3$ und $D \geq 2$ wird sie nur angenommen, wenn $D = 2$ und $d = 3$ (Petersen graph), $d = 7$ (Hoffmann-Singleton Graph, [**6**]) und (vielleicht) falls $d = 57$. Es gibt keine skalierbare Konstruktion der

TABELLE 5.3.1. Graphen mit $m-1=D$. Die Angaben in eckigen Klammern bezeichnen die Vielfachheit eines Eigenwerts.

Graph	$\|V\|$	Grade	Spektrum von L_G	$m-1=D$
Pfad(n)	n	$1,2$	$2-2\cdot\cos(\frac{\pi}{n}j)\ \ j=0,\ldots,n-1$	$n-1$
Kreis(n)	n	2	$2-2\cos\left(\frac{2\pi}{n}j\right)\ \ j=0,\ldots,n-1$	$\lfloor\frac{n}{2}\rfloor$
Star(n)	n	$1,n-1$	$0,1,n$	2
vollst. k-partit(n)	n	$n-\frac{n}{k}$	$0,\deg(G)\,[n-k],,n^{[k-1]}$	2
Clique(n)	n	$n-1$	$0,n\,[n-1]$	1
Hyp(d)	2^d	d	$2j\ \ j=0,\ldots,d$	d
Lattice(k,d)	k^d	$d(k-1)$	$jk\ \ j=0,\ldots,d$	d
Petersen/$d3D2$/MCage(3,5)	10	3	0,2,5	2
Hoff.-Sing./$d7D2$/MCage(7,5)	50	7	0,5,10	2
MCage($d,6$), $d-1$ Primzahlpotenz	$\frac{2(d-1)^3-2}{d-2}$	d	$0,d\pm\sqrt{d-1},2d$	3
MCage($d,8$), $d-1$ Primzahlpotenz	$\frac{2(d-1)^4-2}{d-2}$	d	$0,d\pm\sqrt{2(d-1)},d,2d$	4
MCage($d,12$), $d-1$ Primzahlpotenz	$\frac{2(d-1)^6-2}{d-2}$	d	$0,d\pm\sqrt{3(d-1)},d\pm\sqrt{d-1},d,2d$	6

(d,D)-Graphen. Unter den größten bekannten (d,D)-Graphen haben nur die in Tabelle 5.3.1 angegebenen Graphen einen kleinen Wert von m.

Die (d,t)-*Cage* Netzwerke sind die kleinsten Graphen mit einem gegebenen Grad d und einem kürzesten Kreis der Länge t (Taillenweite) [**6**]. Ein Cage Netzwerk hat mindestens $\frac{d(d-1)^{(t-1)/2}-2}{d-2}$ Knoten für ungerades t und mindestens $\frac{2(d-1)^{t/2}-2}{d-2}$ Knoten für gerades t. Ein Cage Netzwerk, dessen Knotenanzahl mit der unteren Schranke übereinstimmt, wird *minimaler Cage* genannt (MCage(d,t)). Zu diesen Graphen gehören (d,D)-Graphen, die die Moore-Schranke annehmen, Kreise und vollständige bipartite Graphen. Weitere MCage(d,t) Netzwerke existieren nur für $t=6,8,12$ und für Werte von $d-1$, die sich als Primzahlpotenz darstellen lassen. Für MCage Netzwerke gilt $m-1=D=\lfloor\frac{t}{2}\rfloor$. Die Größen der (d,t)-Cage Netzwerke sind nur für eine begrenzte Zahl von Werten für t und d bekannt [**113**]. Es zeigt sich, dass unter den bekannten Cage Netzwerken nur die minimalen Cage Netzwerke kleine Werte für m aufweisen.

Die (d,D)-Graphen und die MCage(d,t) Graphen haben zwar ideale Lastverteilungseigenschaften, sind allerdings nicht für jede beliebige Knotenanzahl konstruierbar. Es gibt jedoch auch skalierbare Graphklassen mit guten Lastverteilungseigenschaften.

Die Knödel Graphen [**83**] können für jede Knotenanzahl konstruiert werden. Ihr Grad ist $\lfloor\log_2 n\rfloor$ und ihr Durchmesser ist höchstens $\lceil\log_2 n\rceil$. Falls n eine Zweierpotenz ist, ist der Durchmesser durch $\lceil\frac{\log_2 n+2}{2}\rceil$ gegeben [**52**]. Dieses ist eine Verbesserung gegenüber den Hypercube Netzwerken. Es konnte gezeigt werden, dass Knödelgraphen der Größe 2^i-2 mit $i>1$ kantensymmetrisch sind [**69**]. Nur bei diesen Knödelgraphen haben die Laplacematrizzen wenig Eigenwerte.

Elsässer, Kràlovič und Monien stellen in [**51**] zwei skalierbare Graphfamilien mit polylogarithmischen Grad und einer polylogarithmischen Anzahl von Eigenwerten vor. Und zwar hat der rekursiv definierte Graph H_n mit n Knoten höchstens den Grad $\log n+5$ und seine Eigenwertanzahl wird von $\log^3 n$ dominiert. Das Produkt aus Grad und Eigenwertanzahl ist zwar bei diesen Graphen asymptotisch geringer als bei Kreis und Clique, allerdings ist es für kleine Graphen größer (es wurden die Graphen mit bis zu 100 Knoten betrachtet). Der Graph $G(n)$ ist eine

Graph G	Dim. d	$\deg(g)$	$m_g - 1$	$c(G,\, d)$
Kreis(64)	1	2	32	64
Clique(64)	1	63	1	63
Torus(8×8)=	1	4	12	48
Kreis(8)^2	2	2	4	16
Hyp(6)=	1	6	6	36
Hyp(2)^3=	3	2	2	12
Hyp(1)^6	6	1	1	6
Lattice(4, 3) =	1	9	3	27
Clique(4)^3	3	3	1	9
Butterfly(4)	1	4	10	40
DeBruijn(6)	1	4	17	68
Knödel(64)	1	6	33	198
Cage(6, 6)	1	6	3	18
Knödel(62)	1	5	7	35
G(64)	1	4	8	32

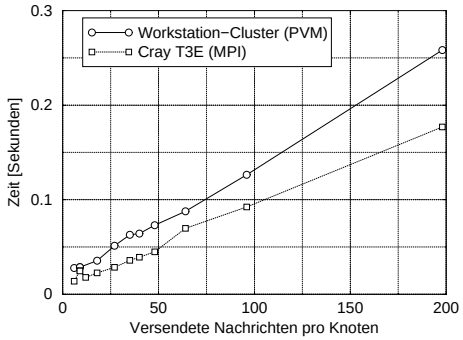


ABBILDUNG 5.3.1. Eigenschaften verschiedener Graphen. Bei kartesischen Produkten haben die Graphen stets die Form $G = g^d$. Die Anzahl Eigenwerte von g wird mit m_g bezeichnet. Es gilt also $c(G, d) = d\,(m_g - 1)\,\deg(g)$. In dem Diagramm sind die Laufzeiten des OPT-IT Algorithmus über die Anzahl der von jedem Knoten versendeten Nachrichten aufgetragen. Die auf der Cray T3E gemessenen Zeiten sind im Diagramm mit 10 multipliziert.

Modifikation einer Hypercube. Sein Grad wird von $\log n$ dominiert und die Anzahl seiner Eigenwerte durch $\log^2 n$. Auch hier ist bei den meisten Graphen mit bis zu 100 Knoten das Produkt aus Grad und Eigenwertanzahl größer als beim Kreis, es stellt sich aber heraus, dass die Instanzen $G(48)$ und $G(64)$ sehr gute Eigenschaften haben (vgl. Tabelle 5.5.2).

Die Tabelle in Abbildung 5.3.1 zeigt einen Vergleich verschiedener Graphen mit jeweils 64 Knoten (außer Cage(6, 6) und Knödel(62) mit 62 Knoten). Die zweite Spalte bezieht sich auf die Anzahl d der Iterationen der äußeren Schleife von OPT-IT. In allen Fällen mit $d > 1$ handelt es sich bei den kartesischen Produkten um Potenzen eines Basisgraphen. Die Spalten drei und vier enthalten den Grad und die Anzahl der Eigenwerte der Basisgraphen. Die fünfte Spalte enthält die Anzahl $c(G, d)$ der Nachrichten, die von OPT-IT pro Knoten maximal generiert werden.

Es gilt beispielsweise Hyp(6) = Hyp(2)^3 = Hyp(1)^6. Je nach dem, ob OPT-IT mit 1, 2 oder 3 Dimensionen durchgeführt wird, erhalten wir jeweils unterschiedliche Werte für $c(G,\, d)$. Und zwar wird $c(G,\, d)$ für größer werdendes d kleiner. Wie wir im nächsten Abschnitt sehen werden, vergrößert sich allerdings in diesem Fall das Flussvolumen. Für $d = 1$ ist der Wert $c(G, d)$ am größten, dafür erhalten wir aber in diesem Fall den l_2-minimalen Fluss.

Das Diagramm in Abbildung 5.3.1 zeigt den linearen Zusammenhang zwischen $c(G, d)$ und der Laufzeit des OPT-IT Algorithmus. Durch lineare Regression erhalten wir

$$T^{(\text{Cluster})}_{\text{OPT}-\text{IT}} \approx 1.2\text{ms} \cdot c(G, d) + 17\text{ms} \;\text{ und }\; T^{(Cray)}_{\text{OPT}-\text{IT}} \approx 0.084\text{ms} \cdot c(G, d) + 0.98\text{ms}\ .$$

5.4. Lastmigration

Nachdem der Fluss berechnet ist, beginnen die Knoten mit der Migrationsphase. Es sei $v \in V$ ein Knoten v mit den Nachbarn $v_1, \ldots, v_k$ im Migrationsgraph $M_G(x)$. Der Knoten muss den Lastbetrag $\tilde{x}_{(v,v_i)}$ an den Nachbarn v_i schicken. Da die initiale Last w_v hierfür eventuell nicht ausreicht, wird die Last gegebenenfalls in mehreren Portionen transportiert. Im einzelnen erfolgt die Migration der Objekte wie folgt.

Die Anzahl der noch an den Nachbarn v_i zu versendenden Lastelemente wird dabei in der Variablen out_{v,v_i} gespeichert, die dementsprechend mit $\tilde{x}_{(v,v_i)}$ initialisiert wird. Anschließend wird out_{v,v_i} jedes Mal, wenn v Lastelemente an v_i schickt, modifiziert. Die Migrationsphase terminiert, sobald $out_{v,w} = 0$ gilt für alle $(v, w) \in \tilde{E}$[1]. Diese Bedingung wird nach einer endlichen Anzahl von Schritten erfüllt, da die von OPT-IT erzeugten Flüsse kreisfrei sind.

Der Verteilungsalgorithmus wird ausgeführt, sobald der Fluss berechnet ist und danach immer dann, wenn Lastelemente von anderen Knoten empfangen worden sind. Es sei $out_v := \sum_{(v,w)\in\tilde{E}} out_{v,w}$ der noch auszuliefernde Fluss von v zu einem dieser Zeitpunkte. Falls out_v nicht größer als die aktuelle Last ist, kann die gesamte auszuliefernde Last in einem Schritt bedient werden. Ein *Konflikt* entsteht immer dann, wenn out_v größer als die momentane Last eines Knotens ist. In diesem Fall muss entschieden werden, wie viele Elemente an welchen Nachbarn versendet werden sollen. Es ist bekannt, dass jeder lokale Greedy-Algorithmus, der bei einem Konflikt seine komplette Last verschickt, höchstens $\sqrt{n}$-mal so viele Schritte benötigt, wie ein optimaler Algorithmus [**44**]. Die hier verwendete *Proportionalheuristik* gehört zu dieser Algorithmenklasse. Sie bewegt den Anteil $out_{v,v_i}/out_v$ der momentan verfügbaren Last an den Knoten v_i. Die Last wird also vorzugsweise in die Richtung der größten Senken bewegt.

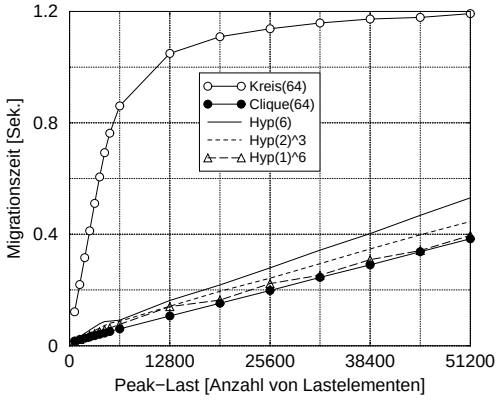
Um den Aufwand der Migration messen zu können, führen wir logische Kommunikationsrunden ein. Wir bezeichnen mit $r(v)$ die Rundennummer von Knoten v. Initial gilt $r(v) = 0$ für alle $v \in V$. Jede Nachricht wird mit der Rundennummer des sendenden Knotens plus eins versehen. Jedes Mal, wenn ein Knoten v eine Nachricht empfängt, setzt er seine Rundennummer auf das Maximum von $r(v)$ und der Rundennummer der Nachricht. Wir definieren die maximale Rundennummer r durch $r := \max_{v\in V} r(v)$.

Bei der Aufwandsabschätzung der Flussberechnungsphase konnten wir die Latenzzeit wegen der kleinen Nachrichtengrößen durch eine Konstante L ausdrücken. Bei der Migrationsphase spielt die Größe der Nachrichten eine entscheidende Rolle. Wir modellieren den Einfluss der Nachrichtengröße, indem wir davon ausgehen, dass eine Nachricht der Größe s eine Latenzzeit von sL verursacht. Wir können somit die Migrationszeit beschränken:

$$T_{\mathrm{m}} \leq r\,(\deg(G)\,o + f(x)\,L) \quad .$$

Die Werte r und $f(x)$ hängen vom Lastfluss ab, der wiederum von der Topologie und des Flussberechnungsverfahrens abhängt. Im Folgenden benutzen wir zwei Lastszenarios um diesen Zusammenhang genauer zu untersuchen. In dem *Peak-Szenario*, setzen wir w_1 auf einen positiven Wert und außerdem $w_i = 0$ für $2 \leq i \leq n$. Das Peak-Szenario modelliert somit sehr unbalancierte Lastsituationen, in denen nur wenige Knoten initial über den Großteil der Last verfügen. Um Anwendungen zu modellieren, die eine balanciertere Ausgangslast haben, betrachten wir das *Zufallsszenario*, in dem wir die Einträge von w zufällig gemäß einer Gleichverteilung bestimmen.

[1] Hier gehen wir von dem idealisierten Fall aus, dass die Lasten beliebig teilbar sind.



G	r	$l_2(x)$	Migrationszeit Cray [s]	Migrationszeit NOW [s]
Kreis(64)	**32**	**35638**	**1.19**	**6.9**
Clique(64)	**1**	**6350**	**0.38**	**2.0**
Hyp(6)=	6	22755	0.40	2.1
Hyp$(2)^3$=	6	32790	0.47	2.1
Hyp$(1)^6$	6	35919	0.53	2.1

ABBILDUNG 5.4.1. Dauer der Migrationsphase für verschiedene Topologien bei Peak-Last. Das Diagramm zeigt den Zusammenhang zwischen der Größe der initialen Last und der Migrationszeit. Die Experimente wurden auf einer Cray T3E durchgeführt. Die nebenstehende Tabelle setzt die Migrationszeit und die l_2-Norm des Flusses in Verbindung.

5.4.1. Das Peak-Szenario. Das Diagramm in Abbildung 5.4.1 zeigt auf der Cray T3E gemessene Migrationszeiten. Bei den Experimenten wurden bis zu 51200 Lastelemente auf Knoten 0 generiert (jedes Lastobjekt besteht aus 150 Bytes). Es wurden fünf Topologien mit jeweils 64 Knoten betrachtet (Kreis, 3 Konstruktionen des Hypercube Netzwerkes und die Clique). Die nebenstehende Tabelle zeigt die Anzahl der Kommunikationsrunden r, die l_2-Norm des Flusses und die Migrationszeit für die Cray und das Clustersystem bei einer initialen Last von 51200 Lastelementen.

In allen Fällen stimmt der Wert für r mit dem Durchmesser des Netzwerkes über ein. Interessanterweise muss dieses nicht notwendigerweise der Fall sein [**44**].

Für eine feste initiale Last ist der Knotenfluss $f(x)$ in allen Fällen gleich. Die lange Dauer der Migrationsphase im Falle der Kreistopologie entsteht durch den großen Durchmesser. Erwartungsgemäß kann die Migrationszeit deutlich verkürzt werden, wenn wir Topologien mit kleinem Durchmesser verwenden.

Die drei Zeiten für die Hypercube beziehen sich auf die drei Möglichkeiten der Flussberechnung (OPT, OPT-IT bzgl. Hyp$(2)^3$ und OPT-IT bzgl. Hyp$(1)^6$). Von den drei Varianten werden die Kanten in unterschiedlichem Maße für die Migration genutzt. Das OPT-Verfahren benutzt alle Kanten der Topologie für die Lastverteilung. Dagegen hängt die Anzahl der Kanten, über die Last migriert wird, bei der OPT-IT Methode der Anzahl der Dimensionen ab. Beispielsweise werden bei sechs Dimensionen nur 63 Kanten, also etwa nur ein Drittel der Kanten, genutzt. Dieser Zusammenhang wird auch an den gemessenen l_2-Normen des Flusses deutlich. Die unbalancierte Verteilung der Kommunikationslast führt bei der Cray zu einer Verlängerung der Migrationszeit. Auf dem Clustersystem ist dieser Effekt nicht zu beobachten, was auf die hohen Aufsetzzeiten zurückzuführen ist. Insgesamt wird die Migrationszeit bei Peak-Last also im wesentlichen vom Durchmesser beeinflusst.

5.4.2. Zufällige Ausgangslast. Der Abstand im Netzwerk zwischen Knoten mit großer und kleiner Last ist bei einer zufällig gewählten Ausgangslast deutlich kleiner als beim Peak-Szenario.

TABELLE 5.4.1. Die durchschnittliche Rundenzahl $\overline{r}$ und die Dauer der Migrationsphase für verschiedene Topologien bei zufällig gewählter initialer Last. Alle Einträge sind Durchschnittswerte von 10 Experimenten bei einer initialen Gesamtlast von 51200 Lastelementen.

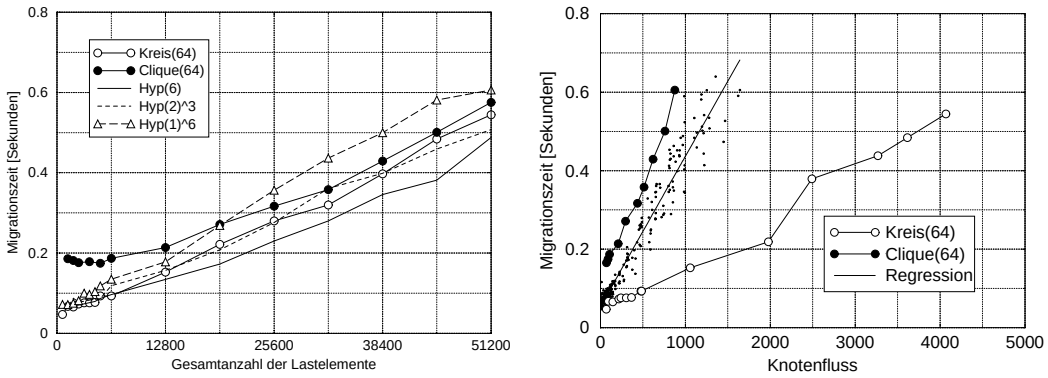
G	$\overline{r}$	$f(x)$	$l_2(x)$	T_m		$T_m/f(x)$		$T_m/l_2(x)$	
				Cray [s]	NOW [s]	Cray [ms]	NOW [ms]	Cray [μs]	NOW [ms]
Kreis(64)	**3.3**	**4067**	**7591**	**0.076**	**0.54**	**0.018**	**0.13**	**9.6**	**0.07**
Clique(64)	**1.0**	**875**	**478**	**0.026**	**0.60**	**0.030**	**0.69**	**53.6**	**1.27**
Torus(8×8)=	1.8	1195	2338	0.024	0.51	0.020	0.43	10.4	0.22
Kreis(8)^2	2.2	1641	3239	0.029	0.61	0.019	0.37	9.6	0.19
Lattice(4,3) =	1.3	929	1345	0.022	0.44	0.024	0.47	16.8	0.32
Clique(4)^3	2.1	1950	1919	0.027	0.60	0.022	0.48	14.4	0.32
Hyp(6)=	2.1	970	1679	0.022	0.49	0.023	0.50	13.6	0.29
Hyp(2)^3 =	2.1	1282	2481	0.025	0.51	0.019	0.40	9.6	0.20
Hyp(1)^6	1.3	1355	2733	0.028	0.64	0.020	0.47	10.4	0.23
Butterfly	2.0	1214	2288	0.024	0.50	0.020	0.41	10.4	0.22
DeBruijn(6)	1.7	1219	2314	0.026	0.54	0.022	0.44	11.2	0.23
Knödel(64)	1.4	991	1376	0.021	0.52	0.022	0.53	12.8	0.31
Knödel(62)	1.4	998	1850	1.022	0.45	0.022	0.45	12.0	0.24
Cage(6, 6)	1.4	1063	1624	0.023	0.49	0.022	0.46	14.4	0.30

Somit ist auch die Anzahl der Migrationsrunden kleiner. Die Tabelle 5.4.1 zeigt die durchschnittliche Anzahl $\overline{r}$ der Migrationsrunden, die bei den unterschiedlichen Topologien benötigt werden. Außer im Falle des Kreises werden durchschnittlich höchstens zwei Runden benötigt. Die Anzahl der Runden hat also im Gegensatz zum Peak-Szenario keinen entscheidenden Einfluss auf die Migrationszeit.

Der dominierende Parameter ist die Dauer einer Migrationsrunde, die wiederum von der zu versendenden Datenmenge, also vom Knotenfluss $f(x)$, abhängt. Abbildung 5.4.2(b) zeigt die deutliche Korrelation zwischen dem Knotenfluss $f(x)$ und der Migrationszeit. Wir können keine perfekte Korrelation erwarten, da die Kostenfunktion T_m auch vom Grad und der Anzahl der Kommunikationsrunden abhängt. Der Knotenfluss ist zu pessimistisch im Falle des Kreises und zu optimistisch im Falle der Clique. Ein Grund hierfür liegt im kleinen Knotengrad des Kreises und im großen Grad der Clique.

Die in Tabelle 5.4.1 aufgelisteten Knotenflüsse zeigen, dass der Knotenfluss sowohl von der Topologie als auch vom Flussberechnungsverfahren abhängt. Je mehr Dimensionen getrennt voneinander behandelt werden, um so größer wird der Knotenfluss.

Um dieses zu verstehen, betrachten wir zum Beispiel die Kreis(8)^2 Topologie. In der ersten Iteration balanciert OPT-IT 8 Kreise gleichzeitig. Im Falle einer zufälligen initialen Lastverteilung ist das System bereits nach dieser ersten Iteration gut balanciert. Die Experimente haben



(a) (b)

ABBILDUNG 5.4.2. Die beiden Diagramme beziehen sich auf die Messungen auf dem Cluster System. Das Diagramm (a) zeigt die Dauer der Migrationsphase in Abhängigkeit der Gesamtsystemlast. Jeder Eintrag ist ein Durchschnittswert von 10 Messungen mit derselben Gesamtlast. Das Diagramm (b) korreliert den Knotenfluss und die Migrationszeit. Es basiert auf 3000 Experimenten, die unter anderem mit den in der Tabelle 5.4.1 gezeigten Topologien durchgeführt worden sind. Jeder Punkt zeigt den durchschnittlichen Knotenfluss sowie die durchschnittliche Migrationszeit von 10 Experimenten mit derselben initialen Gesamtlast.

gezeigt, dass nur etwa 20% des Gesamtflusses über Kanten läuft, die in der zweiten Dimension liegen. Dieses Phänomen hat zwei Konsequenzen. Erstens führt es zu einer unbalancierten Verteilung der Kommunikationslast auf den Kanten. Diesen Effekt haben wir bereits beim Peak-Szenario beobachten können. Zweitens werden potentielle Balancierungspartner nicht in Verbindung gebracht wenn sie sich in unterschiedlichen Dimensionen befinden. Dieses führt wiederum zu überflüssigen Migrationen und somit zu einem hohen Knotenfluss.

Das OPT Verfahren erzeugt kleinere Knotenflüsse, da es alle Kanten gleichberechtigt nutzt und somit unnötige Migrationen vermeidet. Dieser Zusammenhang zwischen der balancierten Kantennutzung (l_2-Norm des Flusses) und der Größe des Kommunikationsvolumens (Knotenfluss) wird auch an den in Tabelle 5.4.1 gezeigten Messungen deutlich. Kleine Werte von $l_2(x)$ implizieren in allen Fällen auch kleine Werte von $f(x)$. Die Diffusionsverfahren scheinen also durch ihren l_2-minimalen Fluss einen Fluss zu erzeugen, der auch stets sehr gut in Bezug auf den Knotenfluss ist. Im Falle der Clique ist der Knotenfluss in der Tat minimal.

LEMMA 5.4.1. *Der eindeutige l_2-minimale balancierende Fluss der Clique mit n Knoten und einem initialen Lastvektor $w \in IR^n$ hat den Knotenfluss $f(x) = \max_{1 \leq i \leq n} |w_i - \tilde{w}| =: \Delta w$.*

BEWEIS. Die Laplacematrix der Clique hat die Eigenwerte 0 und n. Somit berechnet OPT den l_2-minimal Fluss in einer Iteration und der Betrag des Flusses über eine Kante $\{u, v\}$ ist gerade $\frac{|w_u - w_v|}{n}$. Der Knotenfluss eines Knotens v ist also gegeben durch

$$f(v) = \sum_{u \in V} \frac{|w_v - w_u|}{n} \quad .$$

Es seien $w_{\min}$ und $w_{\max}$ die kleinste beziehungsweise die größte initiale Last und es sei v ein beliebiger Knoten. Weiter sei $\underline{V}_v := \{u \in V \mid w_u \leq w_v\}$ und $\overline{V}_v := \{u \in V \mid w_u > w_v\}$. Falls $|\underline{V}_v| < |\overline{V}_v|$ ist, dann gilt

$$\begin{aligned}\sum_{u \in V} |w_v - w_u| &\leq |\underline{V}_v|\,(w_v - w_{\min}) + \sum_{u \in \overline{V}_v} w_u - w_v \\ &\leq \sum_{u \in \overline{V}_v} w_u - w_v + (w_v - w_{\min}) \\ &\leq \sum_{u \in V} w_u - w_{\min}\ .\end{aligned}$$

Andernfalls gilt $\sum_{u \in V} |w_v - w_u| \leq \sum_{u \in V} w_{\max} - w_u$. Folglich ist

$$f(v) = \sum_{u \in V} \frac{|w_v - w_u|}{n} \leq \max\{\sum_{u \in V} \frac{w_u - w_{\min}}{n}, \sum_{u \in V} \frac{w_{\max} - w_u}{n}\} \leq \Delta w.$$

Nun folgt die Aussage des Lemmas aus der Tatsache, dass für mindestens einen der Knoten mit minimaler oder maximaler Last der Knotenfluss gleich Δw ist. □

Der Knotenfluss eines balancierenden Flusses auf einer beliebigen Topologie ist also mindestens so groß, wie der der Clique mit derselben Knotenanzahl (und derselben initialen Last). In diesem Sinne verhält sich die Clique optimal. Wie wir allerdings gesehen haben, ist das Maß des Knotenflusses wegen des hohen Knotengrades der Clique zu optimistisch. Abbildung 5.4.2(a) zeigt, dass bei Verwendung des OPT Verfahrens die Migrationszeit der Clique größer ist als die der Hypercube, obwohl der Knotenfluss der Clique kleiner ist.

Dennoch bleibt als Fazit festzuhalten, dass ein kleiner Knotenfluss die Hauptvoraussetzung für eine kurze Migrationsphase ist, und dass der l_2-minimale Fluss des OPT Verfahrens kleine Knotenflüsse liefert.

5.5. Balancierungskosten

Die Untersuchung der Flussberechnungsphase und der Migrationsphase hat gezeigt, dass sowohl die Wahl der Topologie als auch die des Flussberechnungsverfahrens die Dauer der Lastverteilungsphase beeinflusst.

Für eine schnelle Flussberechnung sollte der Knotengrad klein sein und die Laplacematrix sollte wenig Eigenwerte haben. Falls die Topologie als kartesisches Produkt dargestellt werden kann, kann die Flussberechnung durch Benutzung des OPT-IT Verfahrens beschleunigt werden. Dieses geschieht dann allerdings auf Kosten der Ausgeglichenheit des Flusses. Die Dauer der Migrationsphase hängt von der Lastsituation ab. Wenn die zugrunde liegende Anwendung Lastsituationen erzeugt, die eher mit dem Peak-Szenario zu vergleichen sind, ist es entscheidend, dass die Topologie einen kleinen Durchmesser hat. Wenn sie eher mit einer zufälligen Verteilung der Last zu vergleichen ist, dann sollte der von der Flussberechnung erzeugte Knotenfluss möglichst klein sein.

Es ist somit im Allgemeinen ein Kompromiss zwischen dem Aufwand, der in die Flussberechnung investiert wird und der gesamten Balancierungszeit zu finden. Betrachten wir hierzu als Beispiel die Hypercube mit 64 Knoten. In diesem Fall dauert die Berechnung des Flusses bei Verwendung von OPT-IT mit 6 Dimensionen 0.06 Sekunden auf dem HPCLine-System. Die Berechnung des Flusses mit dem OPT-Verfahren dauert etwa doppelt so lang.

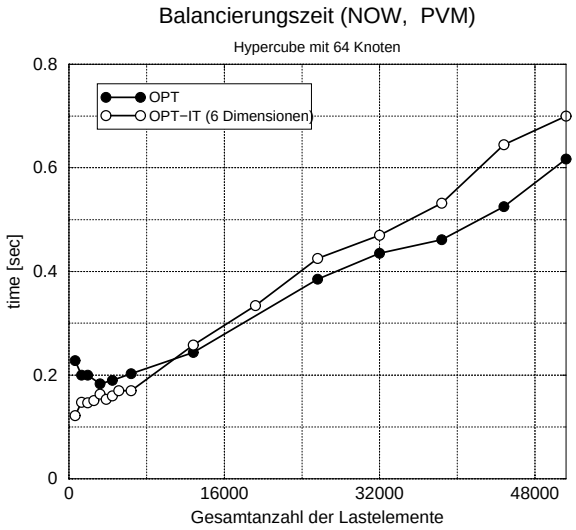


ABBILDUNG 5.5.1. Vergleich der Balancierungszeit von OPT und OPT-IT bei Verwendung eines Hypercube Netzwerks für verschiedene Systemlasten.

Bei einer Durchschnittslast von 10 werden bei Verwendung von OPT-IT mit 6 Dimensionen 0.12 Sekunden für beide Phasen benötigt. Dagegen benötigt OPT 0.23 Sekunden. Die Situation ändert sich bei größeren Durchschnittslasten. Bei einer durchschnittlichen Last von 800 lohnt es sich, den l_2-optimalen Fluss zu berechnen, da hier OPT-IT mit 0.7 Sekunden länger Zeit benötigt, als OPT mit 0.62 Sekunden. Abbildung 5.5.1 veranschaulicht diesen Zusammenhang.

Wenn die Anzahl der zu verteilenden Lastelemente also klein ist, wird es sich nicht lohnen, den l_2-minimalen Fluss zu berechnen. In diesem Fall sollte die Dauer der Flussberechnung mit Hilfe des OPT-IT Verfahrens möglichst gering gehalten werden. Die Wahl einer geeigneten Kombination aus Topologie und Flussberechnungsverfahren ist also anwendungsabhängig und wird daher im Einzelfall durch Experimente zu bestimmen sein.

Die Tabellen 5.5.1 und 5.5.2 zeigen Beispiele regulärer Topologien, deren Laplacematrizzen wenig Eigenwerte haben. Bei den fett gedruckten Topologien ist ihre Anzahl bezüglich der regulären Graphen minimal. Im einzelnen enthält die linke Spalte die Knotenanzahl und die nächsten acht Spalten enthalten die Graphen, deren Knotengrad der Angabe in der Kopfzeile entspricht. Bei den Graphen in der zweiten Spalte handelt es sich also um Graphen, deren maximaler Knotengrad 3 ist. Die rechte Spalte enthält bezüglich der angegebenen Graphen das minimale Produkt aus maximalen Grad und der Anzahl der Eigenwerte sowie in Klammern die Grade, bei denen dieses Produkt erreicht wird. Die Graphen sind in der Form "$m-1[(i)]$, *Bezeichnung, Durchmesser*" notiert. In den Fällen, wo der optionale Wert i angegeben ist, wurden alle nicht isomorphen regulären Graphen wie in [**109**, Kapitel 3] erzeugt. Und zwar bezeichnet i hier jeweils die Anzahl nicht isomorpher Graphen mit einer minimalen Anzahl von Eigenwerten. Die Bezeichnungen sind nach Tabelle 5.5.3 gewählt.

An den Tabellen wird deutlich, dass vor allem für größere Knotenanzahlen nur wenige *gute* Topologien bekannt sind. Die Entwicklung von Graphklassen mit wenigen Eigenwerten und kleinem Grad wie $H(n)$ und $G(n)$ [**51**] sind daher für Lastverteilungszwecke ein Gebiet, auf dem weitere Forschung nötig sein wird.

$\|V\| \setminus d$	3	4	5	6	7	8	9	10	M, (d)
4	**1(1),clique,1**	–	–	–	–	–	–	–	3,(3)
5	–	**1(1),clique,1**	–	–	–	–	–	–	4,(4)
6	**2(1),c2P, 2**	**2(1),c3P,2**	**1(1),clique,1**	–	–	–	–	–	5,(5)
7	–	**3(1),CL,2**	–	**1(1),clique,1**	–	–	–	–	6,(6)
8	**3(1),Hyp3,3**	**2(1),c2P,2**	**3(1),CL,2**	**2(1),c4P,2**	**1(1),clique,1**	–	–	–	7,(7)
9	–	**2(1),T(3^2),2**	–	**2(1),c3P,2**	–	**1(1),clique,1**	–	–	8,(4,8)
10	**2(1),Petersen,2**	**3(2),-,2**	**2(1),c2P,2**	**2(1),-,2**	**3(1),CL,2**	**2(1),c5P,2**	**1,clique,1**	–	6,(3)
11	–	**4(1),-,3**	–	**4(2),-,2**	–	**4(2),-,2**	–	**1(1),clique,1**	10,(10)
12	**4(3),ram(l,3,11),3**	**3(2),-,3**	**3(5),CL,3**	**2(1),c2P,2**	**3(3),CL,2**	**2(1),c3P,2**	**2(1),c4P,2**	**2(1),c6P,2**	11,(11)
13	–	**3(1),CL,2**	–	**2(1),CL,2**	–	**3(1),CL,2**	–	**4(2),-,2**	12,(4,6,12)
14	**3(1),ncage(3,6),3**	**3(1),-,3**	4,CL,3	3,CL,3	**2,c2P,2**	4,CL,2	**3(1),-,2**	**3(1),-,2**	9,(3)
15	–	**3(1),-,3**	–	3,CL,2	–	3,CL,2	–	**2(1),c3P,2**	12,(4)
16	**4(1),-,4**	**4(3),Hyp4,4**	5,CL,4	**2,Lat(2,4),2**	4,CL,2	**2,c2P,2**	3,CL,2	4,CL,2	12,(3,6)
17	–	4,CL,3	–	8,CL,3	–	2,CL,2	–	8,CL,2	16,(4,8,16)
18	**4(1),-,4**	5,ram(l,3,17),3	4,CL,4	5,CL,3	5,CL,2	3,CL,3	**2,c2P,2**	5,CL,2	12,(3)
19	–	9,CL,5	–	3,CL,2	–	9,CL,3	–	9,CL,2	18,(6,18)
20	**5(4),-,4**	6,CL,5	6,CL,5	5,CL,3	6,CL,3	3,CL,2	3,CL,3	**2,c2P2**	15,(3)
21	–	7,CL,3	–	4,CL,3	–	4,CL,2	–	7,CL,2	20,(20)
22	3,T(2 × 11),6	6,CL,5	6,CL,5	11,CL,4	11,CL,3	6,CL,3	6,CL,3	3,CL,3	9,(3)
23	–	11,CL,6	—	11,CL,4	–	11,CL,3	–	11,CL,3	22,(22)
24	7,CCC3,6	5,Bfly3,4	6,CL,3	4,CL,4	5,CL,3	4,CL,3	5,CL,3	4,CL,3	20,(4)
25	–	5,T(5^2),4	–	11,CL,3	–	**2,Lat(2,5),2**	–	3,CL,2	16,(8)
26	13,T(2 × 13),7	**3,ncage(4,6),3**	7,CL,3	5,CL,3	5,CL,3	4,CL,2	4,CL,2	9,CL,2	12,(4)
27	–	8, T(3 × 9), 5	–	3,T(3^3),3	–	4,CL,4	–	7,CL,2	18,(6)
28	14,T(2 × 14),8	5,hbw(28,4,6),4	8,CL,7	6,CL,4	6,CL,3	4,CL,3	5,CL,3	5,CL,2	20,(4)
29	–	7,CL,4	–	14,CL,5	–	7,CL,3	–		28,(28)
30	**4,ncage(3,8),4**	5,hbw(30,4,6),4	8,cage(5,5),3						12,(3)
31	–		–		–		–		30,(30)
32	16,T(2 × 16),9	6,CT(4,3),4	5,Hyp5,5	4,d6D2,2					24,(4,6)

TABELLE 5.5.1. Lastverteilungstopologien mit bis zu 32 Knoten.

5.6. Vergleich von Diffusion und Workstealing bei verteiltem Farming

In diesem Abschnitt betrachten wir das Programmierparadigma des *verteilten Farmings.* Im Gegensatz zum *Farming*, bei dem Aufgaben von einem ausgezeichneten Prozess (dem Farmer) erzeugt werden, generieren beim verteilten Farming alle Prozessoren Last. Im ersten Fall stellt sich die Frage nach der Zuteilung der Aufgaben an die Arbeiterprozesse. Beim verteilten Farming

$\|V\| \backslash d$	3	4	5	6	7	8	9	10	$M_\cdot(d)$
33	–	11,T(3×11),6	–		–		–		32,(32)
34	17,T(2×17),9								33,(33)
35	–	11,T(5×7),5	–		–		–		34,(34)
36	18,T(2×18),10	8,T(6^2),6	9,T($2 \times 3 \times 6$),5	8,T($3^2 \times 4$),4				**2,Lat(2,6),2**	20,(10)
37	–		–		–		–		36,(36)
38	19,T(2×19),10	14,ram(1,3,37),5							37,(37)
39	–	13,T(3×13),7	–		–		–		38,(38)
40	20,T(2×20),11	14,T(5×8),6	11,T($2^3 \times 5$),5						39,(39)
41	–	18,d4D3,3	–		–		–		40,(40)
42	10,CT(3,4),6	15,T(3×14),8	**3,mcage(5,6),3**						15,(5)
43	–		–		–		–		42,(42)
44	22,T(2×22),12	17,T(4×11),7							43,(43)
45	–	14,T(3×15),8	–	8,T($3^2 \times 5$),4	–		–		44,(44)
46	23,T(2×23),12								45,(45)
47	–		–		–		–		46,(46)
48	24,T(2×24),13	9,G(48),8	12,T($2 \times 4 \times 6$),6	9,T(3×4^2),5					36,(4)
49	–	9,T(7^2),6	–		–		–		36,(4)
50	25,T(2×25),13	12,T(5×10),7	6,CT(5,3),4		**2,Hoff-Sing,2**				14,(7)
51	–	17,T(3×17),9	–		–		–		50,(50)
52	26,T(2×26),14	20,T(4×13),8							51,(51)
53	–		–		–		–		52,(52)
54	27,T(2×27),14	17,T(3×18),10	17,T($2 \times 3 \times 9$),6	7,T($3^2 \times 6$),5	7,T(2×3^3),4				42,(6)
55	–	17,T(5×11),7	–		–		–		54,(54)
56	28,T(2×28),15	11,ram(q,3,7),5	15,T($2^3 \times 7$),6						44,(4)
57	–	19,T(3×19),10	–		–		–		56,(56)
58	25,cage(3,9),3,6								57,(57)
59	–		–		–		–		58,(58)
60	30,T(2×30),16	17,ram(1,3,59),6	19,T($2 \times 3 \times 10$),7	17,T($3 \times 4 \times 5$),5					59,(59)
61	–		–		–		–		60,(60)
62	31,T(2×31),16			**3,mcage(6,6),3**					18,(6)
63	–	19,T(7×9),7	–	11,T($3^2 \times 7$),5	–		–		62,(62)
64	15,CCC4,8	8,G(64),8	16,T($2 \times 4 \times 8$),7	6,Hyp6,6			3,Lat(3,4),3		32,(4)

TABELLE 5.5.2. Lastverteilungstopologien mit $33 \leq |V| \leq 64$.

werden die Aufgaben bereits von den Arbeiterprozessen erzeugt und somit ist eher eine Umverteilung notwendig. Dieses Anwendungsmuster stellt eine natürliche Anwendung für die in diesem Kapitel behandelten Lastverteilungsverfahren dar. Anwendungen dieses Paradigmas finden sich zum Beispiel bei der parallelen Tabu-Suche [**13**].

Bezeichnung	Beschreibung
clique	Clique
ckP	Vollständiger k-partiter Graph
ddDD	Kleinster regulärer Graph mit Grad d und Durchmesser D
Hyp$k$	Hypercube der Dimension k
Lat(k, a)	Lattice-Graph der Dimension k über einem Alphabet der Größe a
T$(a \times b)$	$a \times b$-Torus
CCCk	Cube Connected Cycles Netzwerk der Dimension k
Bfly$k$	Butterfly Netzwerk der Dimension k
CL	Zirkularer Graph mit minimaler Eigenwertanzahl [**6**]
CT$(k, l))$	k an den Blättern identifizierte k-äre Bäume mit jeweils l Ebenen
(m)cage(a, b)	(Minimaler) Cage Graph mit Parametern a und b
ram(a, b, c)	Ramanujan Graph mit Parametern a, b und c
hbw(n, d, t)	Graph mit Taillenweite $\geq t$ und größter Bisektionsbreite
G(n)	G-Graph mit n Knoten [**51**]

TABELLE 5.5.3. Bezeichnungen der in den Tabellen 5.5.1 und 5.5.2 aufgeführten Topologien.

VDS stellt für dieses Paradigma im wesentlichen die folgenden Techniken zur Verfügung.

(1) *Work-Stealing.* Work-Stealing ist eine Standardmethode für empfänger-initiierte Lastverteilung. Sobald ein Prozessor keine Aufgaben mehr zu erledigen hat, sendet er eine Anfrage an einen zufällig ausgewählten Prozessor. Sobald ein noch beschäftigter Prozessor eine Anfrage empfängt, teilt er seine Last mit dem anfragenden Prozessor.

(2) *OPT-IT.* Wir werden uns hier auf die Berechnung des l_2-minimalen Flusses bezüglich der Hypercube Topologie beschränken. Die Lastverteilung kann entweder in einer von der Abarbeitung der Aufgaben getrennten Phase erfolgen (*dedizierte Diffusion*) oder parallel zur eigentlichen Abarbeitung (*nebenläufige Diffusion*). Da im zweiten Fall die Aufgabenbearbeitung parallel zur Lastverteilung geschieht, ist eine der Grundvoraussetzungen für die Verwendung des Diffusionsverfahrens verletzt. Die berechneten Flüsse werden also im Allgemeinen nicht mehr balancierend sein.

Abbildung 5.6.1 zeigt einen Vergleich von Work-Stealing, dedizierter Diffusion und nebenläufiger Diffusion. Das Diagramm (a) zeigt, dass die dedizierte Variante weniger effizient als Work-Stealing ist. Dieses liegt an den Wartezeiten während der Flussberechnungsphase. Die nebenläufige Diffusion ermöglicht höhere Effizienzen, da sie die Last balanciert und somit die bei Work-Stealing auftretenden Ineffizienzen gegen Ende der Rechnung verhindert.

Wenn die initiale Last jedes Prozessors so groß ist, dass die Prozessoren während der Flussberechnung ausgelastet sind, dann ist ein balancierender Fluss für die initiale Lastsituation auch balancierend für die Situation nach der Flussberechnung. Dieses ist nicht der Fall, wenn während der Flussberechnung bereits Prozessoren unbeschäftigt sind. In diesem Fall bekommen die unbeschäftigten Prozessoren zu viel Last zugeteilt, so dass sie mehr Zeit für die Bearbeitung ihrer Aufgaben benötigen, als Prozessoren, die während der Flussberechnung beschäftigt sind. Dieser Effekt ist in Abbildung 5.6.1(b) zu sehen. Der Einbruch der Systemauslastung am Anfang der Berechnung hat zur Folge, dass nicht alle Prozessoren gleichzeitig ihre Rechnung beenden. Diese

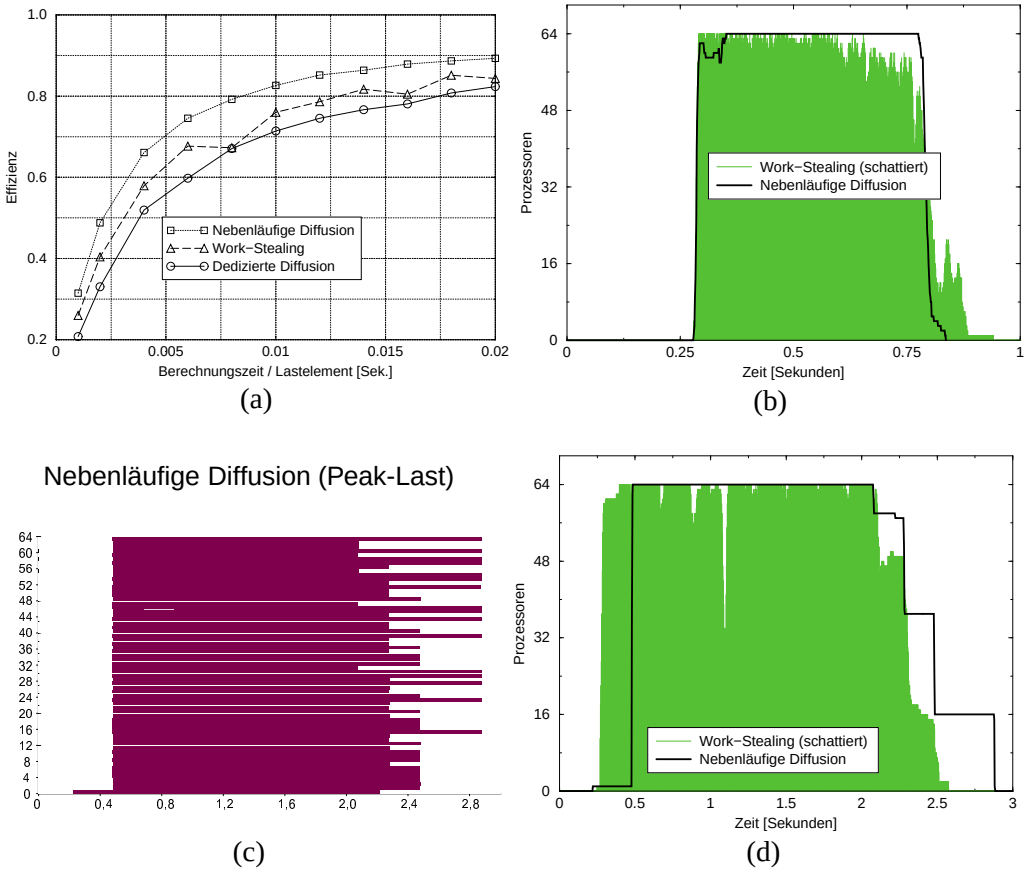


ABBILDUNG 5.6.1. Vergleich von Work-Stealing und Diffusion für verteiltes Farming. Die Experimente wurden auf 64 Knoten des HPCLine-Systems durchgeführt. Für die Diffusionsexperimente wurde das OPT-Verfahren auf einer Hypercube angewendet. Die Diagramme (a) und (b) beziehen sich auf eine zufällige initiale Lastverteilung von 3200 Aufgaben. Das Diagramm (a) setzt die Effizienz der Gesamtberechnung und die Berechnungszeit jeder einzelnen Aufgabe in Beziehung. Jeder Punkt repräsentiert die mittlere Effizienz von 20 Experimenten. Das Diagramm (b) zeigt die Anzahl der beschäftigten Prozessoren bei einer Aufgabenberechnungszeit von 10 Millisekunden. Zur Verdeutlichung ist der Graph unter der Work-Stealing Kurve grau dargestellt. Die Diagramme (c) und (d) stellen die Situation bei einer initialen Peak-Last von 640 Lastelementen und einer Bearbeitungszeit von 0,2 Sekunden dar. Das Ausführungsprofil (c) zeigt die Auslastung der Prozessoren bei Verwendung von nebenläufiger Diffusion.

Ineffizienzen können beispielsweise durch die Initiierung weiterer nebenläufiger Lastbalancierungsphasen oder durch die anschließende Aktivierung von Work-Stealing verringert werden.

Dennoch kann genau dieses Problem dazu führen, dass sich Work-Stealing besser verhält, als nebenläufige Diffusion. Wenn die Last am Anfang sehr unbalanciert ist, entstehen beim Diffusionsverfahren Ineffizienzen einerseits durch Wartezeiten während der Flussberechnung und andererseits dadurch, dass der Fluss bezüglich der Lastsituation nach der Flussberechnung nicht

balancierend ist. Dagegen schafft es Work-Stealing sehr schnell, alle Prozessoren zu beschäftigen. Die Diagramme 5.6.1(c) und (d) illustrieren diesen Zusammenhang an einer Peak-Last Situation. In der in [**13**] betrachteten parallelen Tabu-Suche hat sich in den Experimenten die Work-Stealing Methode als effizienzter erwiesen.

Dieses ist ein Beispiel dafür, dass oft die Wahl eines passenden Lastverteilungsalgorithmus stark anwendungsabhängig ist und mit Hilfe von Experimenten bestimmt werden sollte. VDS unterstützt diese Entwicklungsphase durch die Nutzung geeigneter Standardeinstellungen und durch die Möglichkeit, Lastverteilungsalgorithmen und deren Parameter ohne die Neuübersetzung des parallelen Programms austauschen zu können.

KAPITEL 6

Das Lastverteilungssystem VDS

Die Abkürzung VDS steht für „*Virtual Data* Space“. Dieser Name verkörpert die grundlegende Idee des Systems. Aus der Sicht des Programmierers erzeugt eine VDS-Anwendung Aufgaben, die über einen globalen Datenraum für jeden Prozessor zugreifbar sind. Sobald ein Prozessor bereit ist, Aufgaben zu berechnen, nimmt er sich Aufgaben aus diesem Datenraum. Auf diese Weise wird der Vorgang der Lastverteilung vollständig von der Anwendung entkoppelt. Die Tatsache, dass es sich bei diesem Datenraum eigentlich um verteilte Lastcontainer handelt, deren Last durch Lastverteilungsalgorithmen gesteuert wird, bleibt dem Benutzer (weitgehend) verborgen.

Abschnitt 6.1 gibt einen Überblick über den Stand der Forschung auf dem Gebiet der Lastverteilungsumgebungen. In Abschnitt 6.2 wird das Konzept von VDS beschrieben, und Abschnitt 6.3 beschreibt die in VDS integrierten Lastverteilungsmethoden. Anschließend zeigen die Abschnitte 6.4 und 6.5 die Benutzung von VDS sowie einen experimentellen Vergleich mit drei anderen Lastverteilungsverfahren.

6.1. Stand der Forschung

Die ersten Umgebungen, die dem Anwendungsprogrammierer die Aufgabe der Lastverteilung abnahmen, wurden vor etwa 10 Jahren vorgestellt. Seitdem wurden zahlreiche neue Systeme entwickelt, die sich in den unterstützten Paradigmen und Lastverteilungstechniken oder in den verwendeten Konzepten für die Aufgabenbeschreibung unterscheiden. In der Tabelle 6.1.1 sind einige relevante Lastverteilungsumgebungen aufgelistet. In den folgenden Abschnitten wird näher auf diese Umgebungen und auf ihre Beziehung zu VDS eingegangen.

6.1.1. Unstrukturierte Berechnungen. Kommunikationsbibliotheken wie PVM oder MPI ermöglichen es zwar, kommunizierende Prozesse zu erzeugen, sie stellen aber keine Lastverteilungsmethoden zur Verfügung. Zu den Systemen, die neben der Funktionalität des Nachrichtenaustauschs auch Lastverteilungsfunktionen anbieten, gehören *Chare Kernel* [**120**], *Parallel Objects* (*PO*) [**34**], *CoPA* [**37**], *BALANCE* [**73**] und *PaLaBer* [**108**]. Das von diesen Systemen unterstützte parallele Programmierparadigma ist somit der Nachrichtenaustausch. Sie unterscheiden sich in der Art, in der sie die Aufgaben repräsentieren.

In CoPA, PaLaBer und BALANCE besteht eine parallele Anwendung aus einer Menge kommunizierender Prozesse. Die Systeme zielen darauf ab, verteilte Systeme für grobgranulare Anwendungen zu nutzen. CoPA unterscheidet sich von den anderen beiden Systemen dadurch, dass es die Platzierung der Aufgaben statisch plant. Hierfür muss der Kommunikationsgraph vor der Ausführung bekannt sein. Während der Ausführung sammelt das System Informationen über das Rechen- und Kommunikationsverhalten der Anwendung, die bei der nächsten Platzierung berücksichtigt werden. Für die Planung werden in CoPA die Partitionierungsbibiothek PARTY [**111**]

TABELLE 6.1.1. Lastverteilungsumgebungen. Die erste Spalte enthält die Namen der Systeme, die zweite gibt Auskunft darüber, ob es sich um eine Bibliothek oder eine Programmiersprache handelt, die dritte enthält die Datenstrukturen, mit denen die Aufgaben beschrieben werden, die vierte listet die unterstützten Programmierparadigmen auf und die fünfte Spalte zeigt die von den Systemen verwendeten Lastverteilungsalgorithmen.

System	**Typ**	**Lastbeschr.**	**Paradigmen**	**Lastverteilung**
CHARE	Sprache	Datenstrukt.	aktive Nachrichten	ACWN
Charm++	Sprache	Datenstrukt.	aktive Nachrichten	ACWN
PaLaBer	Bibl.	Prozess	Nachrichtenaustausch	hierarchische Methode
BALANCE	Bibl.	Prozess	Nachrichtenaustausch, Client/Server	zentralisierte Methode
PO	Sprache	Datenstrukt.	Nachrichtenaustausch	Diffusion
CoPA	Bibl.	Prozess	Nachrichtenaustausch	Partitionierung, SA
Dynamo	Bibl.	Datenstrukt.	unabhängige Aufgaben	—
Cilk	Sprache	C-Funktion	strikte Baumberechnung globale Variablen	Work-Stealing
PM^2	Bibl.	C-Funktion	Fork-Join, unterbrechendes Scheduling	verteilte Methode mit globaler Kommunikation
DTS	Bibl.	C-Funktion	Fork-Join	Work-Stealing
DOTS	Bibl.	C++-Meth.	Fork-Join	zentralisierte Methode
RAPID	Sprache	C-Funktion	DAG-Planung	Planungsheuristiken
PYRROS	Sprache	C-Funktion	DAG-Planung	Planungsheuristiken
VDS	Bibl.	Datenstrukt.	Nachrichtenaustausch, (gewichtete) unabhängige Aufgaben, strikte Baumberechnung, DAG-Planung	Diffusion, Dimension Exchange, Work-Stealing, Pie-Mapping, ETF
Mentat	Sprache	C++-Meth.	datengetriebene Berechnung	Graphpartitionierung
Athapascan-1	Sprache	C++-Meth.	datengetriebene Berechnung	Work-Stealing, Planungsheuristiken
ICC++	Sprache	C++-Meth.	datengetriebene Berechnung	initiale Datenplat–zierung, Work-Stealing
ParList	Bibl.	Feld	Gebietszerlegung	lokale Austauschheur.
Dome	Bibl.	Feld	Gebietszerlegung	lokale Austauschheur.
PadFEM	Bibl.	FEM-Netz	Gebietszerlegung	Diffusion
Millipede	Sprache	C-Funktion	globale Variablen	zentralisierte Methode

und Simulated Annealing (SA) benutzt. BALANCE und Pa-La-Ber nehmen die Platzierungsentscheidungen zur Laufzeit vor. BALANCE nutzt hierfür einen zentralisierten und Pa-La-Ber einen hierarchisch organisierten Lastverteilungsalgorithmus.

In der Programmiersprache Chare und in der PO Umgebung werden die Aufgaben durch Datenstrukturen repräsentiert. Ein *Chare* besteht aus einer Reihe von Aktivierungspunkten. Alle Berechnungen werden durch das Senden von Botschaften an diese Aktivierungspunkte initiiert.

Der Kernel startet die Berechnung, indem er Initialisierungsbotschaften an jeden neuen Chare schickt. Da Chares sehr einfach migriert werden können, dann das System auch für feingranulare Berechnungen genutzt werden. Es wird bei der Lastverteilung angestrebt, kommunizierende Chares nahe beieinander zu halten. Um dieses zu erreichen, wird der lokale ACWN Algorithmus (Adaptive Contracting Within Neighborhood) [**136**] angewendet. Die Charm++ Sprache bietet ein objektorientiertes Interface zum Chare Kernel [**81**] an.

Die PO Umgebung basiert auf dem *Modell der aktiven Objekte* [**20**]: jedes PO Objekt verfügt neben der üblichen Objektstruktur aus Zustand und Methoden über eventuell mehrere ausführende Einheiten (Threads). Mindestens ein Thread ist mit jedem PO Objekt verbunden. Es können weitere Threads erzeugt werden, um parallele Berechnungen auf den Objektdaten durchzuführen. Allerdings können diese Threads ausschließlich auf die Daten „ihres“ Objekts zugreifen. Wenn Dienste von anderen Objekten benötigt werden, werden Aufträge an diese Objekte verschickt. Um hierbei Wartezeiten zu vermeiden, unterstützt PO asynchrone Aufträge. PO versucht, wie der Chare Kernel, durch lokale Lastverteilung kommunizierende Objekte auf benachbarten Prozessoren zu halten. In PO wird hierfür ein auf Diffusion basierender Ansatz genutzt [**33**].

Ein Lastverteilungssystem, das nicht direkt das Paradigma des Nachrichtenaustauschs unterstützt, ist die Dynamo Bibliothek [**123**]. Wie bei Chare und PO werden in Dynamo die Aufgaben durch Datenstrukturen repräsentiert. Dynamo erlaubt zusätzlich die Definition von Aufgabenprioritäten. Die Bibliothek bietet allerdings lediglich eine Schnittstelle für Lastverteilungsverfahren an. Die Verfahren selbst müssen vom Benutzer zur Verfügung gestellt werden.

6.1.2. Strikte Baumberechnungen. Die erste Programmierumgebung, die das Paradigma der strikten Baumberechnung (Definition 1.1.1) unterstützte, ist die Programmiersprache Cilk [**8**]. Cilk erweitert die C-Sprache durch eine Reihe von Schlüsselwörtern für das Erzeugen und Synchronisieren von Threads sowie für die Beschreibung der Datenabhängigkeiten. Die Lastverteilung erfolgt durch Work-Stealing [**9**]. Im Gegensatz zu den anderen hier beschriebenen Umgebungen ist Cilk allerdings ausschließlich auf Systemen mit gemeinsamen Speicher lauffähig.

Das Fork-Join Paradigma besteht aus strikten Baumberechnungen, die höchstens eine Datenabhängigkeit zwischen zwei Threads erlauben. In einem *Fork*-Schritt erzeugt die Anwendung neue Threads, und in einem *Join*-Schritt wartet sie auf Ergebnisse (einiger) erzeugter Threads.

Die Bibliotheken DTS [**14**], der objektorientierte Nachfolger DOTS [**7**] und PM^2[**103**] erlauben die Implementierung von Fork-Join Algorithmen. Es gibt dabei zwei wesentliche Unterschiede zwischen der PM^2 Bibliothek und den Bibliotheken DTS und DOTS. Zum einen führt die PM^2 Bibliothek alle erzeugten Threads quasiparallel aus, indem Kontextumschaltungen durch Zeitscheiben realisiert werden. Im Gegensatz hierzu starten DTS und DOTS einen neuen Thread nur dann, wenn der erzeugende Thread durch einen Join-Schritt blockiert ist. Zum anderen erlaubt PM^2 dem Benutzer die Zuweisung von Prioritäten an die Aufgaben. Implementiert werden die Prioritäten durch unterschiedliche Zeitscheibengrößen.

6.1.3. Andere Anwendungsstrukturen. Wenn der Abhängigkeitsgraph der Aufgaben vor der Berechnung bekannt ist, können statische Planungssysteme wie RAPID [**56**] oder PYRROS [**63**] benutzt werden. RAPID nutzt DTS [**139**] und in PYRROS wird das DSC Verfahren [**140**] eingesetzt, um effiziente Pläne vor der Ausführung zu berechnen. Für den Fall, dass der Abhängigkeitsgraph nicht im voraus bekannt ist, wurden die Systeme Mentat [**66**], Athapascan-1 [**59**] oder Illinois Concert C++ (ICC++) [**19, 18**] entwickelt. Durch die Implementierung eines datengetriebenen Ausführungsmodells können diese Systeme beliebige Aufgabengraphen zur Laufzeit

planen. In Mentat wird der Aufgabengraph in regelmäßigen Zeitabständen neu partitioniert und umverteilt [**134**]. Athapascan-1 wendet sowohl dynamische Techniken wie Work-Stealing als auch statische Techniken an. Im letzteren Fall wird von Athapascan-1 das PYRROS System verwendet. ICC++ benutzt eine priorisierte Work-Stealing Technik und Heuristiken für die initiale Platzierung von Aufgaben. Bei allen drei Systemen wird bei der Lastverteilung angestrebt, in Beziehung stehende Aufgaben auf demselben Prozessor zu halten.

Datenabhängigkeiten spielen auch bei datenparallelen Anwendungen eine wesentliche Rolle. Beispiele solcher Anwendungen sind adaptive numerische Simulationen. Systeme wie ParList [**53**], Dome [**2**] oder PadFEM [**45**] bieten parallele Datenstrukturen an, die zu Beginn der Rechnung partitioniert und verteilt werden. Nach jeder Berechnungsrunde werden die Partitionen neu geformt um auf Änderungen der Datenstruktur oder der zur Verfügung stehenden Rechenleistung reagieren zu können. Ein anderer Ansatz derartige Systeme zu realisieren, ist die Simulation gemeinsamen Speichers. Ein Beispiel ist das *Millipede* System, das unter anderem eine verteilte Plattform für ParC-Programme bietet [**54**]. Für die Lastverteilung wird in Millipede eine zentralisierte Methode eingesetzt.

6.1.4. Innovationen und Beschränkungen von VDS. Die wesentlichen Leistungsmerkmale der VDS-Bibliothek sind die effiziente Ausführung strikter Baumberechnungen auf Systemen mit verteiltem Speicher, die Handhabung von Aufgabenprioritäten und die Integration von statischen und dynamischen Lastverteilungsverfahren in einer Bibliothek. Zudem ist VDS das erste Lastverteilungssystem, dass die gleichzeitige Nutzung mehrerer Paradigmen ermöglicht. Auf der anderen Seite unterstützt VDS keine allgemeinen datengetriebenen Berechnungen. Weiterhin berücksichtigt VDS keine Abhängigkeiten zwischen den Aufgaben. Somit ist VDS nicht für Anwendungen geeignet, die auf Gebietszerlegung basieren oder gemeinsamen Speicher benötigen. VDS konkurriert also nicht mit Systemen wie Athapascan-1, PadFEM, oder Millipede.

- Im Gegensatz zu PM^2, DTS und DOTS implementiert VDS das vollständige Modell der strikten Baumberechnungen. Dadurch können generierte Threads mehrmals Ergebnisse generieren. Im Gegensatz zu Cilk kann VDS auf allen Systemen benutzt werden, die PVM oder MPI unterstützen.
- VDS beinhaltet effiziente Verfahren für die Balancierung unstrukturierter Anwendungen. VDS erlaubt wie Dynamo und PM^2 die Definition von Aufgabenprioritäten. VDS benutzt dabei unter anderem die maximale Priorität eines Objekts als Lastmaß. Durch die Verwendung dieses Lastmaßes balanciert VDS sowohl die Anzahl als auch die Priorität der Aufgaben. Aufgaben mit hoher Priorität werden also vor denen mit niedriger Priorität bearbeitet. Dieses Verhalten ist besonders für die Implementierung von verteiltem Best-First Branch&Bound wichtig, da es den Suchoverhead reduziert. Im Gegensatz hierzu generiert der Prioritätsmechanismus von PM^2 bei dieser Anwendung systematisch Overhead, da jedes Teilproblem eine gewisse Berechnungszeit zugeteilt wird [**41**]. Somit investiert PM^2 Rechenzeit in Teilprobleme, die vom VDS nicht ausgewertet werden würden.
- Viele auf dynamischer Programmierung beruhende Anwendungen generieren reguläre Aufgabengraphen, wie sie in Kapitel 3 behandelt worden sind. Insbesondere diese Graphen sind keine strikte Baumberechnungen und werden somit nicht von PM^2, DTS, DOTS, und Cilk unterstützt. Im Gegensatz zu RAPID und PYRROS erlaubt VDS die Integration von optimierten Planungsverfahren für Graphklassen. Ein Beispiel hierfür

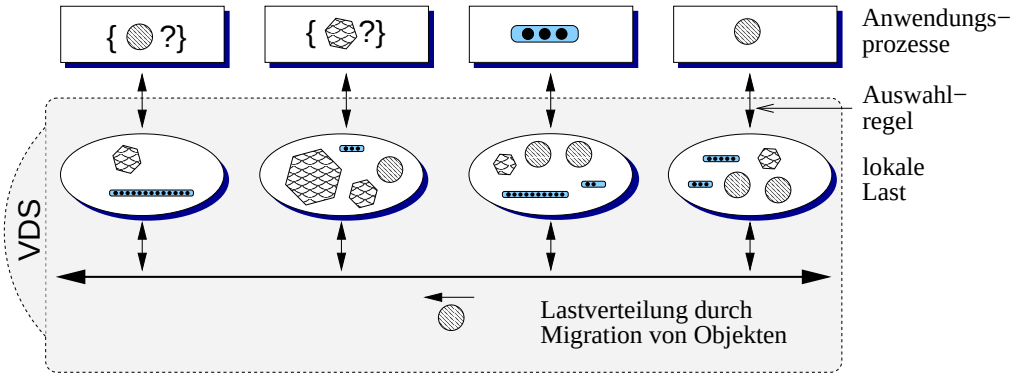


ABBILDUNG 6.2.1. Die Struktur von VDS.

ist die Pie-Mapping Technik für (umgekehrte) Pyramiden und Diamanten. Für beliebige Graphen verwendet VDS den Approximationsalgorithmus ETF [**74**].

- Die Systeme in Tabelle 6.1.1 sind außer VDS auf spezielle Programmierparadigmen abgestimmt. Diese Beschränkung kann unter Umständen ein Hinderniss für effiziente Parallelisierungen sein. Die Descartes Methode für reelle Nullstellenisolierung ist zum Beispiel ein Divide&Conquer Algorithmus, der die Berechnung von Polynomtransformationen in jedem Divide-Schritt erfordert. Diese Transformation geschieht durch die Auswertung einer Pyramide. Collins u.a. haben eine Parallelisierung dieses Algorithmus mit dem DTS System vorgenommen [**28**]. Hierdurch konnten sie nur die Baumparallelität ausnutzen. Sie kamen somit zu dem Ergebnis, dass der Algorithmus nur sehr wenig Parallelität bietet. Für Polynome mit zufällig gewählten Koeffizienten war der Speedup in ihren Experimenten nie größer als 2. Im nächsten Kapitel werden wir sehen, dass wir mit VDS eine skalierbare Parallelisierung erhalten, wenn wir dynamische Lastverteilung auf der Baumebene mit statischer Lastverteilung auf der Knotenebene kombinieren.

VDS bietet dem Benutzer unterschiedliche Lastverteilungstechniken an, mit denen die Verwaltung des globalen Aufgabenraumes der Anwendung angepasst werden kann. Darüberhinaus erlaubt es VDS, anwendungsspezifische Lastverteilungsverfahren und Datenstrukturen zu integrieren. Hierdurch eignet sich VDS auch sehr gut als Testumgebung für neue Lastverteilungs- und Planungsalgorithmen.

Die folgenden Abschnitte gehen auf die Anwendungsschnittstelle, auf die integrierten Lastverteilungsverfahren und auf die Schnittstelle, die VDS für die Integration neuer Komponenten anbietet, ein.

6.2. Die Anwendungsschnittstelle

Die Anwendungsschnittstelle von VDS besteht aus allgemeinen Funktionen und aus Funktionen, die bestimmte Programmierparadigmen unterstützen. Wir beginnen die Beschreibung der Schnittstelle mit den allgemeinen Funktionen. Danach gehen wir näher auf die einzelnen Paradigmen sowie auf die Kombination der Paradigmen ein.

6.2.1. Die Benutzersicht. Eines der grundlegenden Konzepte von VDS ist die Beschreibung der Aufgaben durch Datenstrukturen (*Objekte*). Um eine Anwendung mit VDS zu parallelisieren

ist es somit notwendig, unabhängige Aufgaben zu identifizieren und geeignete Datenstrukturen für sie zu entwerfen. Eine typische VDS Anwendung fordert Objekte von VDS an, bearbeitet sie und generiert neue Objekte. VDS ist für die Verwaltung dieser Objekte verantwortlich. Ein trivialer Ansatz, die Objektverwaltung zu realisieren besteht darin, alle Objekte auf einem Prozessor (Server) zu speichern. Dieser Ansatz würde allerdings Kommunikationskosten bei jeder Objektanfrage zur Folge haben. Außerdem ist dieser Ansatz nicht skalierbar, da der Server schnell zu einem Flaschenhals werden würde. Es ist daher notwendig, die Aufgaben verteilt zu speichern. Aus Effizienzgründen versteckt VDS diese Tatsache nicht vor dem Benutzer.

Verteilte Lastcontainer. Die Abbildung 6.2.1 zeigt die interne Struktur von VDS. Auf jedem Prozessor befindet sich ein Anwendungsprozess und ein Container mit lokal verfügbaren Objekten. Der Anwendungsprozess wird ausschließlich mit Objekten aus diesem Container bedient. Wenn er leer ist, dann ist es die Aufgabe des Lastverteilungsalgorithmus, Aufgaben aus anderen Containern zu transferieren. Wenn also die Anwendung Objekte mit Prioritäten benutzt, dann wird VDS das Objekt mit der höchsten Priorität in Bezug auf die Objekte im Container liefern (und nicht ein Objekt mit systemweit höchster Priorität).

Sobald ein neues Objekt generiert ist, bleibt es unter der Kontrolle von VDS bis es der Anwendung nach einer Lastanforderung zurückgegeben wird. Solange ein Objekt der Kontrolle von VDS unterliegt, kann es beliebig migriert werden. Es wird dabei kein Versuch unternommen, den Aufenthaltsort der Objekte zu verfolgen. Es ist daher nicht möglich, ein *spezielles* Objekt von VDS anzufordern (in LINDA [**15**] ist dieses beispielsweise möglich). Weiterhin geht VDS davon aus, dass es keine besondere Eignung bestimmter Anwendungsprozesse für die Bearbeitung einer Aufgabe gibt. Jedes Objekt sollte von jedem Anwendungsprozess gleich gut bearbeitet werden können.

Lastanforderung. Es gibt zwei Möglichkeiten, Objekte von VDS anzufordern—die blockierende und die nicht blockierende Lastanfrage. Da sich beide Anfragearten ausschließlich auf den lokalen Lastcontainer beziehen, schlägt die nicht blockierende Anfrage eventuell fehl (selbst dann, wenn noch Objekte in anderen Containern vorhanden sind). Die blockierende Anfrage wartet, bis ein Objekt in dem lokalen Container verfügbar ist. Um zu verhindern, dass auf diese Weise eine Verklemmung entsteht, müssen Anwendungen, die nicht selbst das Ende der Berechnung erkennen, die Terminierungserkennung von VDS aktivieren.

Objekttypen und Objektklassen. Die VDS-Objekte werden auf zwei Weisen klassifiziert. Die übergeordnete Klassifikation bezieht sich auf das Programmierparadigma. Im einzelnen sieht VDS für die Unterscheidung der Paradigmen die sechs in Tabelle 6.2.1 aufgeführten *Objekttypen* vor. Der Typ *Aufgabe* wird für die Beschreibung unabhängiger Aufgaben benutzt. Unabhängige Aufgaben, die mit Prioritäten versehen sind, werden durch *gewichtete Aufgaben* repräsentiert. Sie unterscheiden sich von ungewichteten Aufgaben neben der Reihenfolge in der sie bearbeitet werden darin, dass es möglich ist, eine untere Schranke für das Gewicht festzulegen. Objekte, deren Gewicht unterhalb dieser Schranke liegt, werden gelöscht. Diese Funktionalität ist insbesondere für die Parallelisierung von Best-First Branch&Bound von Nutzen. Die Benutzung der *Thread Objekte* wird in Abschnitt 6.2.2 und die der *DAG-Objekte* in Abschnitt 6.2.3 beschrieben. Die *Nachrichtentypen* sind nicht nur für die Kontrolle der parallelen Anwendung hilfreich, sie machen auch die Anwendung unabhängig vom benutzten Kommunikationssystem (PVM oder MPI).

Neben den Objekttypen können Objekte auch anhand der *Klasse*, zu der sie gehören, unterschieden werden. Alle Objekte, die zu derselben Klasse gehören, haben auch denselben Typ.

TABELLE 6.2.1. Der Typ eines Objekts bestimmt die Operationen, die auf das Objekt angewendet werden können. Ferner legte er die Anfangsauswahl der Datenstruktur und der Lastverteilung fest.

Typ	**Paradigma**	**Default Datenstruktur**	**Default Lastvert.**
Aufgabe	unabhängige Aufgaben	FIFO	adaptives Work-Stealing
gewichtete Aufgabe	unabhängige Aufgaben mit Prioritäten	Prioritätswarteschlange	lokale Austauschmethode
Thread	strikte Baumberechnung	Deque für die Aufgaben	Work-Stealing
DAG-Objekt	gerichtete kreisfreie Aufgabengraphen	DAG	ETF (statische Planung)
Botschaft	Nachrichtenaustausch zwischen Anwendungsprozessen	FIFO	keine Lastverteilung
priorisierte Botschaft	Nachrichtenaustausch mit Prioritäten	Prioritätswarteschlange	keine Lastverteilung

Sowohl die Datenstruktur für die Container als auch der Lastverteilungsalgorithmus sind Klassenparameter. Zwei Klassen desselben Typs können also von unterschiedlichen Lastverteilungsalgorithmen verwaltet werden.

Statische und dynamische Objekte. VDS sieht eine statische und eine dynamische Methode für die Speicherung von Objektdaten vor. Die Speichermethode ist ebenfalls ein Klassenparameter. Statische Objekte beinhalten eine feste Datenstruktur, deren Größe bei ihrer Erzeugung angegeben wird. Dynamische Objekte können beliebige Datenstrukturen enthalten. Sie sind zwar flexibler als statische, haben dafür allerdings den Nachteil eines erhöhten Aufwands bei ihrer Migration. Da die Migration durch Nachrichtenaustausch vorgenommen wird, müssen die Datenstrukturen dynamischer Objekte vor ihrem Transport in eindimensionale Datenfelder verpackt werden. Das Packen, Entpacken und das Löschen dynamisch gespeicherter Objekte erfolgt durch Rückruffunktionen, die bei der Klassendeklaration spezifiziert werden.

Ein Beispiel. Die Abbildung 6.2.2 zeigt ein kurzes Beispiel für die Verwendung des Farming Paradigmas. Jeder Anwendungsprozess führt dasselbe Programm aus. Es verwendet zwei Objektklassen. Die erste Klasse (`job`) wird für die Verteilung der Aufgaben benutzt. Die zweite Klasse dient zur Rückgabe der Ergebnisse an den Farmer. Jede VDS Anwendung besteht aus einer *Konfigurierungs-* und einer *Berechnungsphase*. In der ersten Phase (Zeilen 1 und 2) werden die Klassen definiert und verschiedene weitere Einstellungen, zum Beispiel an den Lastverteilungsmethoden, vorgenommen. Die Funktion `VDS_Configure` beendet die Konfigurierungsphase und initialisiert alle Teile von VDS, die für die Anwendung benötigt werden.

In den Zeilen 4 und 5 generiert der Farmer (Prozessor 0) mit der Funktion `VDS_Generate` Aufgaben. Wenn die Aufgaben zu anderen Knoten migriert werden, werden sie automatisch mit den in Zeile 1 spezifizierten Funktionen `pack_fn` und `unpack_fn` gepackt und wieder entpackt. Nachdem alle Aufgaben generiert sind, wird auch der Farmer zum Arbeiter und fordert Aufgaben von VDS an.

```
1  job = VDS_Define_class ("job", VDS_Task, pack_fn, unpack_fn, free_fn);
2  result = VDS_Define_class ("result", VDS_Message, NULL, NULL, NULL);
3  VDS_Configure ();
4  if (VDS_id == 0) for (i=0; i<n; i++)
5     VDS_Generate (VDS_New_object (job, sizeof (sub_problem), input+i));
6  classes = VDS_Class_list (2, job, result);
7  VDS_Activate_term_detection (classes);
8  while ((class_id = VDS_Wait (classes, &obj)) != -1) {
9     if (class_id == job) {
10       x = process (VDS_Object_data (obj));
11       VDS_Place (VDS_New_message (result, sizeof (int), &x), 0);}
12    else
13       combine (X, VDS_Object_data (obj));
14    VDS_Delete_object (obj);}
```

FIGURE 6.2.2. Ein einfaches Farming-Beispiel.

In Zeile 6 wird eine Liste von Klassen erzeugt, die für die Berechnung relevant sind. Die Reihenfolge der Aufgaben in dieser Liste entspricht der Reihenfolge, in der VDS die Objekte im lokalen Lastcontainer durchsucht. In diesem Beispiel überprüft VDS zunächst, ob Objekte der Klasse `job` vorhanden sind. Falls dieses nicht der Fall ist, wird die Klasse `result` überprüft. Die Zeile 7 stellt sicher, dass die Berechnung stoppt, sobald alle Aufgaben berechnet sind. Die eigentliche Berechnung der Aufgaben beginnt in Zeile 8.

Solange noch unbearbeitete Aufgaben im System vorhanden sind, fordern die Prozesse durch den Aufruf von `VDS_Wait` Aufgaben von VDS an. Sobald die Terminierungserkennung festgestellt hat, dass keine Aufgaben mehr vorhanden sind, liefert die Funktion den Wert -1. Andernfalls liefert sie ein Objekt. Wenn es ein Objekt der Klasse `job` ist, wird es bearbeitet, und das Ergebnis wird mit `VDS_Place` an den Prozessor 0 geschickt. Wenn es ein `result`-Objekt ist, wird es benutzt um das endgültige Ergebnis zu konstruieren.

Das resultierende VDS-Programm kompensiert unterschiedliche Rechenkapazitäten der Prozessoren sowie unterschiedliche Bearbeitungszeiten der Aufgaben. Durch die Nutzung der adaptiven Work-Stealing Methode [**40**] wird der Lastverteilungsaufwand automatisch der Granularität der Anwendung angepasst.

Die folgenden Abschnitte beschreiben die VDS Unterstützung strikter Baumberechnungen und statischer Aufgabengraphen und führen das Konzept der dynamischen Prozessorgruppen ein.

6.2.2. Strikte Baumberechnungen. Die Schnittstelle für strikte Baumberechnungen umfasst Funktionen für das Erzeugen von Threads und für die Handhabung von Ergebnissen. Die Abbildung 6.2.3 zeigt ein Beispielprogramm, das Fibonaccizahlen berechnet.[1]

Das Konzept für die Übermittlung eines Ergebnisses vom Kind-Thread zum Vater-Thread basiert auf dem speziellen Ergebnis-Typ `VDS_Result`. Die Threads deklarieren Variablen dieses Typen als einen Teil ihrer Datenstruktur. Beispielsweise erzeugt der erste Aufruf von `VDS_Fork` ein Thread-Objekt für die Berechnung der $(n-1)$-ten Fibonaccizahl und verbindet das Ergebnis dieses Objekts mit der Ergebnisvariablen `f1` des Vater-Threads. Jedes Mal, wenn ein Kind

[1]Dieses Proramm nutzt einen ineffizienten Algorithmus, der exponentielle Zeit benötigt. Obwohl logarithmische Methoden bekannt sind [**32**], ist das Programm dennoch ein gutes Beispiel für strikte Baumberechnungen.

```
#include <vds.h>
#include <forkjoin.h>

typedef struct {
               int         n;
               int         step;
              VDS_Result  f1, f2; } fib_args;

void fibonacci (fib_args *args) {
  int n = args->n;
  fib_args child;

  child.step=0;

  if (n <= 1)
    VDS_Return(sizeof (int), &n);
  else
    switch (++args->step)
      {
      case 1 :  {child.n = n-1;
                 VDS_Fork(sizeof (fib_args), &child, &(args->f1)); break;}
      case 2 :  {child.n = n-2;
                 VDS_Fork(sizeof (fib_args), &child, &(args->f2)); break;}
      case 3 :  {VDS_Join(2, &(args->f1), &(args->f2)); break;}
      case 4 : {int fib =*(int *)VDS_Access (&(args->f1)) +
                         *(int *)VDS_Access (&(args->f2));
                 VDS_Return(sizeof (int), &fib);}
      }
  }

int main (int argc, char **argv)

  {
  int          *fib;
  VDS_ClassId   fib_thread;
  fib_args      args;

  VDS_Init (&argc, &argv))
  fib_thread = VDS_Define_class ("fib", VDS_Thread, NULL, NULL, NULL)
  VDS_Declare_handler (fib_thread, fibonacci);
  VDS_Configure();

  args.step = 0;
  args.n    = atoi (argv[1])
  fib = VDS_ForkJoin (fib_thread, &args, sizeof (args));
  if (VDS_id == 0) printf (" fib(%d) = %d\n", args.n, *fib);

  VDS_Finalize ();
  }
```

ABBILDUNG 6.2.3. Berechnung der n-ten Fibonaccizahl mit VDS.

ein Ergebnis generiert, wird es zu der Variable transferiert, die für dieses Ergebnis bestimmt ist. Da Vater- und Kind-Thread ihre gegenseitigen Aufenthaltsorte im System nicht kennen, benötigt VDS im schlimmsten Fall drei Nachrichten für die Übermittlung des Ergebnisses. Da aber die Work-Stealing Methode üblicherweise zusammenhängende Teile des Aufrufbaumes auf demselben Prozessor hält, geschieht die Ergebnisübermittlung in den meisten Fällen ohne Kommunikation.

Nach dem Aufruf von `VDS_Fork` wird die Kontrolle über beide Threads an VDS übergeben. Die nächste Lastanfrage wird mit dem neu erzeugten Objekt bedient. In der Zwischenzeit kann der Vater-Thread eventuell von einem anderen Prozessor gestohlen werden und dort einen weiteren Kind-Thread erzeugen. Blumhofe und Leiserson haben gezeigt, dass diese Planungsmethode jede strikte Baumberechnung in asymptotisch optimaler Zeit berechnet [**9**]. In manchen Fällen kann es allerdings von Vorteil sein, mehrere Kind-Threads ohne Unterbrechung erzeugen zu können. Die Funktion `VDS_Spawn` bietet diese Möglichkeit.

Der Kind-Thread erzeugt sein Ergebnis mit der Funktion `VDS_Return`. Bevor der Vater-Thread auf Ergebnisse seiner Kind-Threads zugreifen kann, muss er die Ergebnisse durch einen Aufruf der Funktion `VDS_Join` anfordern. In Schritt 4 des Algorithmus sind somit die Ergebnisse beider Kind-Threads verfügbar und können mit `VDS_Access` abgefragt werden.

Das Fibonacci-Beispiel demonstriert die Nutzung von *Bearbeitungsroutinen*. Während der Berechnung übernimmt VDS die Programmkontrolle. Solange wie noch Objekte zu verarbeiten sind, entnimmt VDS Objekte aus den Containern und verarbeitet sie mit Hilfe der vom Benutzer definierten Bearbeitungsroutinen. Hier wird beispielsweise die `fibonacci` Funktion als Bearbeitungsroutine für Objekte der Klasse `fib_thread` deklariert. Für die parallele Berechnung einer Fibonaccizahl, führt jeder Prozessor die Funktion `VDS_ForkJoin` aus. Das Endergebnis wird als Funktionsergebnis auf Prozessor 0 generiert.

Allgemeine strikte Baumberechnungen erlauben den Kind-Threads die Übermittlung mehrerer Ergebnisse zu unterschiedlichen Zeitpunkten. Dieses ist nicht möglich, wenn `VDS_Fork` benutzt wird, da diese Funktion nur die Angabe einer Ergebnisvariablen erlaubt. VDS bietet jedoch auch Funktionen, die auf Listen von Ergebnissen operieren und somit die Implementierung allgemeiner strikter Baumberechnungen erlauben.

6.2.3. Gerichtete kreisfreie Aufgabengraphen. VDS sieht die Integration von Planungsalgorithmen für Graphfamilien und beliebige Aufgabengraphen vor. Die Angabe, welche Graphfamilie oder welchen Graphen eine Anwendung nutzt, muss während er Konfigurationsphase erfolgen. Eine Graphfamilie wird durch die Deklaration eines geeigneten Planungsalgorithmus definiert. Die genaue Größe des Graphen (wie zum Beispiel die Höhe einer Pyramide) muss erst unmittelbar vor seiner Berechnung angegeben werden und kann für jede Instanz andere Werte annehmen. Dieses ist bei beliebigen Graphen nicht möglich, da sie während der Konfigurierungsphase spezifiziert werden müssen.

Die Idee der VDS Schnittstelle für statische Aufgabengraphen ist, sowohl die Kanten als auch die Knoten des Graphen als VDS-Objekte zu repräsentieren. Die Anwendung verarbeitet Knoten und erzeugt Kanten. Zu Anfang werden Kanten generiert, die die Eingabekanten des Graphen darstellen. In Abbildung 3.1.1 (Seite 22) sind diese Kanten außerhalb der grauen Zonen dargestellt. Der Lastverteilungsalgorithmus für Aufgabengraphen verteilt diese Kanten gemäß des angegebenen Planungsalgorithmus und setzt aus den Kanten neue Knoten zusammen. Sobald alle Eingabekanten eines Knotens vorhanden sind, kann er berechnet werden. Es sei angemerkt, dass die Berechnung von Aufgabengraphen eine Forderung an die Objekte stellt, die eigentlich dem

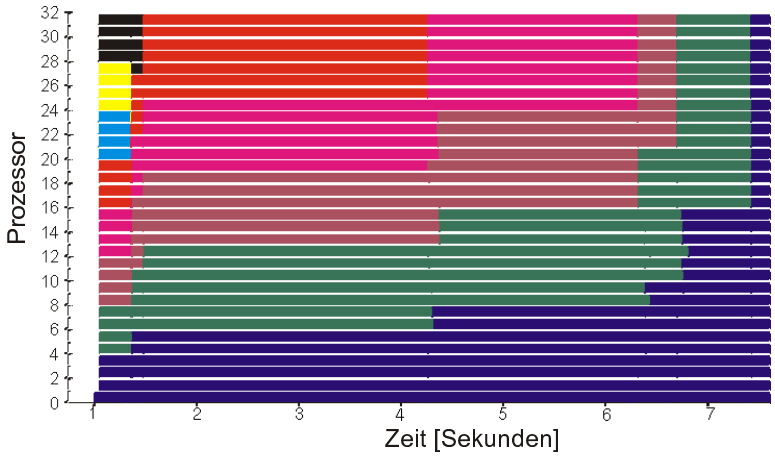


ABBILDUNG 6.2.4. Die Gruppenstruktur von 8 parallelen Nullstellenisolierungen (Kapitel 7). Die horizontale Achse zeigt die Zeit, und die vertikal angeordneten Streifen zeigen die Gruppen, zu denen die Prozessoren gehören. In jeder Gruppe wird eine Nullstellenisolierung durchgeführt. Jedes Mal, wenn eine Isolierung terminiert, wird die zugehörige Gruppe gelöscht und die Prozessoren werden an andere Gruppen verteilt.

Konzept von VDS widerspricht. Die Objekte müssen in diesem Fall nämlich *adressierbar* sein. Da sich aber die Verteilung der Knoten nach einem statischen Plan richtet, ist dieses tolerierbar.

Die erste Eingabekante eines Graphen wird mit der Funktion `VDS_New_DAG` generiert. Diese Funktion informiert VDS über wichtige Parameter des Graphen. Zu diesen gehören die Größe (im Falle von Graphfamilien) und die Anzahl der Prozessoren, die für die Berechnung genutzt werden sollen. `VDS_New_DAG` erzeugt einen eindeutigen Bezeichner für den Graph, der an alle Kanten und Knoten übergeben wird. Auf diese Weise ist es möglich, mehrere Graphen gleichzeitig auszuwerten.

6.2.4. Dynamische Prozessorgruppen. Das Kapitel 4 hat sich Anwendungen gewidmet, die aus mehreren Parallelitätsebenen bestehen. Dort standen allerdings Anwendungen im Vordergrund, die parallele Aufgaben generieren, deren Ausführungszeit erstens bekannt und zweitens gleich ist. Falls nichts über die Ausführungszeit bekannt ist, bietet VDS die Möglichkeit, die Prozessoren rekursiv in disjunkte *Gruppen* einzuteilen. Für jede dieser Gruppen wird ein *Größenparameter* angegeben. Die Anzahl der Prozessoren, die jeweils einer Gruppe zugeteilt werden, ist proportional zu diesen Parametern. Jede Gruppe wird durch ihren Gruppenbezeichner eindeutig identifiziert. Dieser Bezeichner wird wiederum allen Objekten, die von der entsprechenden Gruppe bearbeitet werden sollen, zugewiesen. Sobald die Berechnung in einer Untergruppe terminiert, wird die entsprechende Gruppe gelöscht und die frei gewordenen Prozessoren werden unter Berücksichtigung der Größenparameter an Brudergruppen verteilt. Die Abbildung 6.2.4 illustriert diesen Prozess.

6.2.5. Topologien. Wie wir in Kapitel 5 gesehen haben, benutzen Lastbalancierungsalgorithmen häufig feste Topologien für die Festlegung der Kommunikationspartner. Hierbei sind ein kleiner Grad, ein kleiner Durchmesser und eine möglichst geringe Anzahl m von Eigenwerten wichtig. VDS unterstützt die Basistopologien Kreis, Clique, DeBruijn, Butterfly und minimale

TABELLE 6.3.1. In VDS integrierte Lastverteilungsmethoden.

Methode	**Anwendung**	**Last-beobachtung**
Scattering	Lastteilung	—
Zufällige Platzierung	Lastteilung	—
Work-Stealing	Lastteilung	—
Adaptives Work-Stealing	Lastteilung	lokale Last
Diffusion (FOS)	Lastbalancierung	Lasttabelle
Dimension Exchange	Lastbalancierung	Lasttabelle
Dimension Exchange (rand.)	Lastbalancierung	Lasttabelle
Lokaler Austausch	Lastbalancierung	Lasttabelle
Flussplanung	Lastbalancierung	Flusstabelle
Qualitätsbalancierung	Lastbalancierung für gewichtete Aufgaben	Lasttabelle
Baumbalancierung	Lastteilung für Threads	zentral
Pie-mapping	Planung von DP-DAGs (Abb. 3.1.1)	—
ETF	Planung von allgemeinen DAGs	—

Cages mit 14, 26, 42 und 62 Knoten. Ferner erlaubt VDS die Bildung von Produkten dieser Graphen, wodurch auch Tori, Hypercubes und Lattice-Graphen erzeugt werden können. Per Default wählt VDS aus diesen Graphen eine Topologie mit minimalem Produkt aus maximalem Grad und m (vgl. Tabellen 5.3.1, 5.5.1 und 5.5.2).

6.3. Lastverteilungsmethoden

Es ist eine Vielzahl von Lastverteilungsmethoden für unterschiedliche Anwendungsfelder vorgestellt worden. Eine gute Übersicht findet sich beispielsweise in [**117**]. Es scheint schwierig, vielleicht sogar unmöglich zu sein, einen Algorithmus anzugeben, der für alle Anwendungen geeignet ist. Die Wahl einer geeigneten Strategie hängt von Anwendungsparametern, wie zum Beispiel der Balancierungsqualität oder der Art der Lasterzeugung (vgl. Abschnitt 5.6), sowie von Hardwareparametern ab.

VDS bietet deswegen ein ganzes Spektrum von Lastverteilungsmethoden an (vgl. Tabelle 6.3.1). Der Anwendungsprogrammierer kann aus diesen Methoden die geeignete Methode für seine Anwendung auswählen. Darüberhinaus kann er eigene Methoden integrieren.

Die Klassen werden unabhängig voneinander balanciert. Der Lastverteilungsalgorithmus für eine gegebene Klasse wird immer dann aufgerufen, wenn ein neues Objekt generiert worden ist, wenn die Lastbeobachtung die Notwendigkeit von Lastbalancierungsschritten entdeckt hat, oder wenn ein Prozessor keine Aufgaben mehr zu bearbeiten hat.

In den nächsten Abschnitten folgt eine kurze Beschreibung der in Tabelle 6.3.1 aufgelisteten Verfahren. Sie beginnt mit einem wichtigen Unteralgorithmus, der Lastbeobachtung.

6.3.1. Lastbeobachtung. Die Lastbeobachtung hat die Aufgabe, das dynamische Verhalten der Anwendung zu überwachen und nötigenfalls den Lastverteiler aufzurufen. Dieses ist normalerweise dann der Fall, wenn größere Lastschwankungen aufgetreten sind. Die *Last* wird stets in Bezug auf eine bestimmte Klasse gemessen. Die Methode, mit der die Last bestimmt wird, ist ein

Klassenparameter. Es stehen die folgenden Lastmaße zur Verfügung. (Die letzten beiden Maße sind nur für gewichtete Objekte definiert.)

- Die Anzahl lokal gespeicherter Objekte,
- die Anzahl lokal gespeicherter Objekte, multipliziert mit der exponentiell geglätteten durchschnittlichen Objektbearbeitungszeit,
- das größte lokale Objektgewicht und
- die l_2-Norm der Objektgewichte.

Die einfachste Form der Lastbeobachtung betrifft nur den Prozessor selbst (lokale Lastbeobachtung). In diesem Fall wird der Lastverteilungsalgorithmus aktiviert, sobald sich die lokale Last um einen bestimmten Faktor geändert hat.

Wenn jedoch die Last *balanciert* werden soll, dann ist es notwendig, auch die Last anderer Prozessoren zu beobachten. In diesem Fall versorgt die Lastbeobachtung den Lastverteilungsalgorithmus mit einer Tabelle, die Prozessorindizes und entsprechende Lastinformationen enthält. Diese Lastinformation ist entweder die aktuelle Last eines Prozessors *(Lasttabelle*) oder sie entspricht einer Lastmenge, die zu dem entsprechenden Prozessor geschickt werden soll (*Flusstabelle)*.

Synchrone Lasttabellen. Eine synchrone Lasttabelle enthält Lastinformationen des jeweiligen Prozessors sowie seiner Nachbarn in der Topologie. Der Aktualisierungsprozess der Tabelle ist wie beim Algorithmus 5 (OPT-IT) in Runden organisiert. Am Anfang jeder Runde schickt jeder Prozessor seine Last an alle Nachbarn. Der Lastverteilungsalgorithmus wird aktiviert, sobald alle Lastinformationen der Nachbarn eingetroffen sind. Nach einem bestimmten Zeitintervall wird die nächste Runde gestartet. Je kürzer diese Zeit ist, desto genauer ist die Lastbeobachtung. Diese Beobachtungsmethode eignet sich sehr gut für das FOS Verfahren. Sie hat allerdings den Nachteil, dass der Beobachtungsaufwand nicht adaptiv zur Anwendung ist.

Adaptive Lasttabellen. Dieser Algorithmus stellt der Anwendung einen adaptiven Aktualisierungsprozess zur Verfügung. Die Nachbarn eines Prozessors werden nur dann über die lokale Last informiert, wenn diese sich um einen bestimmten Faktor geändert hat. Der Lastverteilungsalgorithmus wird aktiviert, wenn der relative Lastunterschied zu einem Nachbarprozessor ein gewisses Maß überschreitet. Experimente mit einem parallelen Branch&Bound Algorithmus für das TSP haben gezeigt, dass sogar sehr große Systeme mit bis zu 1024 Prozessoren mit dieser Methode effizient genutzt werden können [**126**].

Randomisierte Lasttabellen. Als dritte Alternative bietet VDS die in [**40**] vorgestellte Technik an. Es ist eine zeitgesteuerte Variante des von Lüling und Monien analysierten Lastverteilungsalgorithmus [**97**]. Die Prozessoren senden ihre Lastinformationen in regelmäßigen Zeitabschnitten an zufällige Prozessoren. Alle empfangenen Lastinformationen werden in der Lasttabelle gespeichert. Nach einer bestimmten Anzahl von Zeitschritten aktivieren alle Prozessoren, die mindestens eine Lastinformation empfangen haben, ihren Lastverteiler. Anschließend wird der Inhalt der Lasttabelle gelöscht. Im Gegensatz zur synchronen und zur adaptiven Lasttabelle ändern sich also hier von Runde zu Runde die Balancierungspartner. Die in [**40**] durchgeführten Experimente haben gezeigt, dass diese Methode in heterogenen Clustersystemen der adaptiven Lasttabelle überlegen ist.

Flussberechnung durch Diffusion. Für die Berechnung von Flusstabellen benutzt VDS das in Kapitel 5 diskutierte OPT-IT Verfahren.

6.3.2. Dynamische Lastverteilung. VDS enthält eine Reihe von Lastteilungs- und Lastbalancierungsverfahren, von denen einige für bestimmte Objekttypen spezialisiert sind (Tabelle 6.3.1). Im Folgenden gehen wir kurz auf die integrierten Verfahren ein.

Lastteilungsmethoden. Eine einfache aber oft sehr nützliche Methode ist das *Scattering*-Verfahren [**137**]. Es ist eine senderinitiierte Methode, die neu generierte Aufgaben, falls die lokale Last einen bestimmten Grenzwert überschreitet, reihum verteilt. Eine Variante hiervon platziert die neuen Objekte auf zufällig ausgewählten Prozessoren (*zufällige Platzierung*). Beispielsweise kann Farming mit diesen Methoden gut parallelisiert werden.

Ein sehr effektives empfängerinitiiertes Verfahren ist *Work-Stealing.* Falls die Last eines Prozessors unter einen bestimmten Grenzwert U fällt, versucht er, Last von anderen Prozessoren zu stehlen. Für strikte Baumberechnungsverfahren haben Blumhofe und Leiserson gezeigt, dass Work-Stealing mit $U = 0$ asymptotisch optimal ist [**9**]. Per Standardeinstellung stehlen die Prozessoren *ein* Objekt bei Verwendung des Objekttyps Thread und die *Hälfte* der Last bei normalen oder gewichteten Aufgaben.

Bei normalen oder gewichteten Aufgaben ist es oft möglich, die für die Bearbeitung der sich im lokalen Lastcontainer befindlichen Objekte benötigte Rechenzeit abzuschätzen. Um eine Abschätzung der Rechenzeit zu bekommen, kann VDS die Berechnungszeiten für die Objekte einer Klasse verfolgen. Durch eine exponentielle Glättung erhält man auf diese Weise einen guten Schätzwert für die verbleibende Rechenzeit. Das *adaptive Work-Stealing* Verfahren ist eine Lastteilungstechnik, die diese Abschätzung nutzt. Hier werden Stehlversuche dann unternommen, wenn die erwartete Restrechenzeit unter einen bestimmten Wert fällt. Der Vorteil dieser Methode ist, dass sie die Granularität der Anwendung berücksichtigt wird, und dass durch die Wahl der Lastschranke die für den Stehlvorgang benötigte Zeit kompensiert werden kann.

Alle sender- und empfängerinitiierten Verfahren können kombiniert werden. Auf diese Weise kann ein gemischt-initiiertes Verhalten erzeugt werden.

Für strikte Baumberechnungen, deren Aufgaben parallel verarbeitet werden können, steht die in Abschnitt 4.4 diskutierte Methode, die für jede Baumebene phasenparallele Pläne benutzt, zur Verfügung (*Baumbalancierung*). Allerdings muss für die Verwendung dieser Methode das Laufzeitverhalten der Aufgaben bekannt und gleich sein. Andernfalls können dynamische Prozessorgruppen für die Verteilung der Prozessoren an die Baumknoten benutzt werden.

Lastbalancierungsmethoden. Alle Lastbalancierungsverfahren benutzen Lasttabellen oder Flusstabellen. Die Balancierungspartner sind jeweils die in der Tabelle enthaltenen Prozessoren. Die Techniken unterscheiden sich vor allem darin, wie sie die Last an diese Partner verteilen.

Die Verfahren, die Lasttabellen benutzen, balancieren die Last kontinuierlich, wogegen die auf Flusstabellen basierende Lastverteilung in Runden abläuft—sobald der Fluss berechnet ist, wird die Last gemäß des Flusses verteilt. Nach der Verteilungsphase können neue Balancierungsrunden begonnen werden.

Für die kontinuierliche Lastverteilung benutzt VDS zum einen Standardverfahren, wie das FOS, Dimension Exchange und die von Willebeek-LeMair und Reeves vorgeschlagene lokale Austauschmethode [**135**]. Zusätzlich zu diesen Verfahren stellt VDS eine randomisierte Variante des Dimension Exchange Verfahrens, bei der die Balancierungspartner in einer zufälligen Reihenfolge behandelt werden, zur Verfügung (Algorithmus 6). Diese Methode wurde im Rahmen der *Paderborn Parallel Branch&Bound-Library* entwickelt und hat sich bei verteiltem Branch&Bound als sehr gut erwiesen [**124, 125**]. Durch Änderung des Schwellwerts δ kann die Lastbalancierungsqualität den Anforderungen der Anwendung angepasst werden.

Algorithmus 6 Ein randomisiertes Dimension Exchange Verfahren.

Eingabe: lokale Last l, Anzahl der Nachbarn n, Last der Nachbarn $l_1, \ldots, l_n$, Grenzwert δ.

Ausgabe: Anzahl der Lastelemente m_i, die zum Nachbar i geschickt werden sollen.

1. $\pi \leftarrow$ Zufällige Permutation von n Elementen
2. **for** $i \leftarrow 1$ **to** n **do**
3. **if** $\frac{l-l_{\pi(i)}}{l} > \delta$ **then**
4. $m_{\pi(i)} \leftarrow \frac{l-l_{\pi(i)}}{2}$
5. $l \leftarrow l - m_{\pi(i)}$
6. **else**
7. $m_{\pi(i) \leftarrow 0}$

Für gewichtete Aufgaben benutzt VDS standardmäßig einen der obigen Lastbalancierungsalgorithmen um die Anzahl der Objekte zu balancieren. Zusätzlich wird ein lokaler Austausch vorgenommen, um das Gewicht der Objekte zu balancieren. Anders als bei Algorithmus 6 wird bei einem deutlichen Lastunterschied nur ein Objekt (das mit dem zweitgrößten Gewicht) migriert.

Für die Lastverteilung gemäß eines vorher berechneten Flusses benutzt VDS die in Kapitel 5 diskutierte Proportionalheuristik [**44**].

6.3.3. Statische Planung. In der derzeitigen Version von VDS sind lediglich Planungsverfahren integriert, die jeder Aufgabe genau einen Prozessor zuweisen. Die Schnittstelle für die Integration von Planungsverfahren ermöglicht jedoch auch die Integration von redundanten Verfahren, die Aufgaben unter Umständen mehrfach auf unterschiedlichen Prozessoren berechnen, um Kommunikation zu sparen.

Neben dem ETF-Verfahren für allgemeine Aufgabengraphen stellt VDS für die in Kapitel 3 diskutierten Graphklassen das Pie-Mapping Verfahren mit Ebenen- und Blockplanung zur Verfügung. Die Aufgaben einer umgekehrten Pyramide werden in der umgekehrten Reihenfolge wie die der Pyramide abgearbeitet. Gemäß Lemma 3.2.4 erhalten wir somit einen Plan, bei dem dieselbe Zeit für die Kommunikation zur Verfügung steht, wie beim ursprünglichen Pie-Mapping. Für den Diamanten werden die Pläne für eine Pyramide und eine umgekehrte Pyramide, wie in Abbildung 6.3.1 dargestellt, kombiniert.

6.4. Parametrisierung und Modifikation von VDS

Die meisten in VDS integrierten Lastverteilungsverfahren können durch einen oder mehrere Parameter an die Anwendung angepasst werden. Die Werte dieser Parameter können im Quellcode der Anwendung festgelegt, beim Programmstart durch Kommandozeilenparameter angegeben oder in einer Konfigurationsdatei gesetzt werden. Die letzten beiden Methoden haben den Vorteil, dass sie ohne Neuübersetzung eine Änderung der Parameter erlauben. Eine Konfigurationsdatei könnte beispielsweise wie folgt aussehen.

```
tree_node.LOAD_BALANCER=TREE_BALANCER
refine_interval.LOAD_BALANCER=ADAPTIVE_WORKSTEALING
refine_interval.LB_MIN_WORK=0.8
```

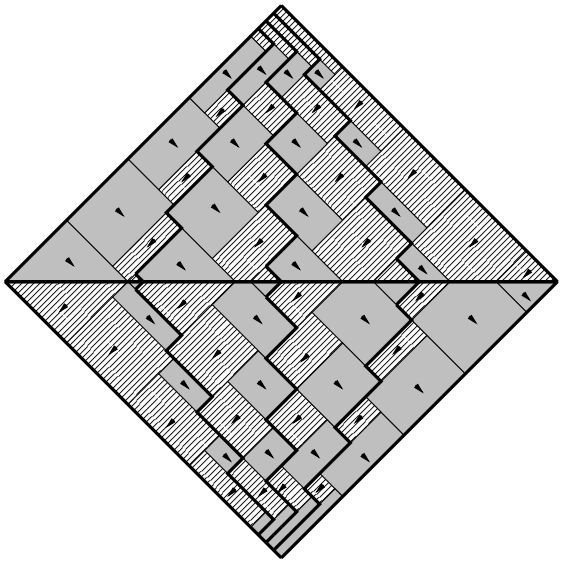


ABBILDUNG 6.3.1. Pie-Mapping mit Blockplanung für den Diamanten. Dargestellt ist die Planung eines Diamanten mit 36×36 Knoten für 4 Prozessoren.

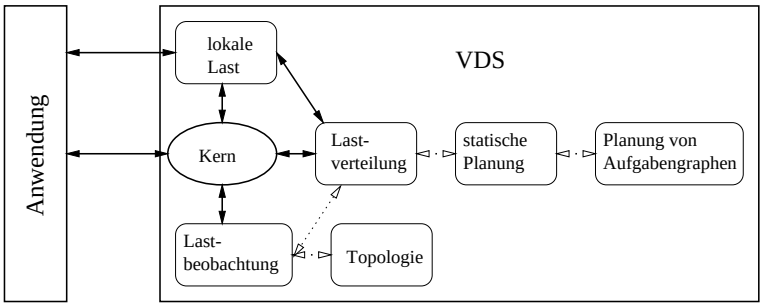


ABBILDUNG 6.4.1. Die interne Struktur von VDS.

Bei dieser Konfiguration wird die Baumbalancierungsmethode für die Objekte der Klasse `tree_node` verwendet. Objekte der Klasse `refine_interval` werden durch adaptives Work-Stealing verteilt. Stehlversuche werden initiiert, sobald die erwartete verbleibende Rechenzeit weniger als 0.8 Sekunden beträgt.

Neben der Parametrisierung der integrierten Verfahren kann das Verhalten von VDS auch durch die Integration neuer Verfahren, zum Beispiel für die Lastverteilung, verändert werden. Abbildung 6.4.1 zeigt, wie die einzelnen Module von VDS in Verbindung stehen. Mit Ausnahme des Kerns enthält VDS für jedes dieser Module mehrere Implementierungen. Jede Implementierung stellt die für das entsprechende Modul erforderlichen Funktionen zur Verfügung. Zum Beispiel besteht jeder Lastverteilungsalgorithmus aus den folgenden Funktionen.

- `init` – wird von VDS einmal für jede Klasse aufgerufen.
- `place` – wird jedes Mal aufgerufen, wenn ein neues Objekt entweder von der Anwendung generiert wird oder von einem anderen Prozessor empfangen wird. Der Lastverteiler muss entscheiden, ob dieses Objekt im lokalen Container gespeichert oder zu einem

anderen Prozessor migriert werden soll. Beispielsweise speichert Work-Stealing alle neuen Objekte lokal.

- `acquire` – wird aufgerufen, wenn eine Lastanfrage nicht mit den lokal verfügbaren Objekten bedient werden kann. In diesem Fall initiiert Work-Stealing, wenn nicht bereits geschehen, Stehlversuche.
- `load_changed` – wird vom Lastbeobachtungsmodul aufgerufen (falls es aktiviert ist). Die adaptive Work-Stealing überprüft hier die verbleibende Rechenzeit. Falls sie kürzer als der vorgesehene Grenzwert ist, wird ein Stehlversuch gestartet.

Ein neuer Lastverteilungsalgorithmus wird durch den Aufruf von

```
id = VDS_Register_load_balancer( name, descr,
                                 my_init, my_place, my_acquire, my_load_ch,
                                 types, preferred_load_table)
```

integriert. Der Parameter `name` wird in der Konfigurationsdatei benutzt, um den neuen Lastverteiler für eine Klasse auszuwählen. `descr` ist eine kurze Beschreibung der verwendeten Lastverteilungsmethode, der `types`-Parameter gibt die Objekttypen an, für die dieser Lastverteiler benutzt werden kann, und der letzte Parameter wählt die standardmäßig benutzte Lastbeobachtungsmethode aus. Das Ergebnis der Funktion ist ein neuer Bezeichner, der im Programm benutzt werden kann, um die Nutzung dieses Lastverteilers für eine bestimmte Klasse zu bewirken. Auf ähnliche Weise ist es—ohne Veränderung von VDS—möglich, neue Datenstrukturen für die Container, neue Lastbeobachtungsverfahren, andere Topologien, neue Planungsverfahren für Graphfamilien oder neue Algorithmen für die Planung beliebiger Aufgabengraphen zu integrieren.

6.5. VDS in der Praxis

Die VDS Bibliothek ist in diversen Anwendungen, die sich innerhalb des Sonderforschungsbereichs 376 “Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen” ergeben haben, erfolgreich eingesetzt worden. An den in Tabelle 6.5.1 aufgelisteten Anwendungen wird deutlich, dass durch die Unterstützung verschiedener Paradigmen und Lastverteilungsalgorithmen für sehr unterschiedliche Anwendungstypen effiziente Parallelisierungen möglich sind. Die Parallelisierung der Descartes Methode für polynomielle Nullstellenisolierung ist ein Beispiel für die Kombination zweier Paradigmen. Sie wird im nächsten Kapitel behandelt.

Dieser Abschnitt zeigt einen Vergleich von VDS mit Cilk und DOTS (Abschnitt 6.1.2) sowie mit Athapascan-1 (Abschnitt 6.1.3). Alle vier Systeme erlauben die Parallelisierung von Anwendungen, die das Fork-Join Paradigma benutzen. Ein Beispiel einer solchen Anwendung ist die in Abschnitt 6.2.2 diskutierte rekursive Berechnung von Fibonaccizahlen. Das vollständige VDS-Programm ist in Abbildung 6.2.3 dargestellt. Abbildung 6.5.1 zeigt die Hauptroutinen der Implementierungen mit den drei anderen Systemen.

Athapascan-1 ist eine C++ Bibliothek, die durch Makros eine Datenfluss-Sprache implementiert. Der Datenflussgaph wird durch globale Variablen definiert. Die Parameter der Funktionen greifen entweder lesend oder schreibend auf diese Variablen zu. Eine Funktion, die lesend auf Variablen zugreift, wird ausführbar, sobald andere Funktionen für die Variablen Werte definiert haben. Somit wird die `sum`-Funktion im Fibonacci Beispiel nur dann ausgeführt, wenn die beiden Summanden definierte Werte haben.

Cilk ist eine prozedurale Sprache für strikte Baumberechnungen. Sie fügt der C-Sprache spezielle Schlüsselwörter für die Erzeugung und die Synchronisierung von Threads hinzu. Nachdem

- Athapascan-1:

```
class fibo: public a1_task {
   public:
     void operator()( a1_value< int > n) {
        if( n.val() <= 1 ) {
            c.write( new int( n.val() ));
         } else {
            a1_shared_rp_wp< int > a( 0 );
            a1_shared_rp_wp< int > b( 0 );
            a1_fork( fibo(), (a1_value< int >) (n.val()-1), a );
            a1_fork( fibo(), (a1_value< int >) (n.val()-2), b );
            a1_fork( sum(), a, b, c );
         }
      }
   };
```

- Cilk:

```
cilk int fib(int n,) {
     if (n <= 1)
       return (n);
     else {
       int x, y;
        x = spawn fib(n - 1);
        y = spawn fib(n - 2);
       sync;
       return (x + y);
      }
   }
```

- DOTS:

```
int* fib(int *n) {
     int *result, *resultA, *resultB;

      result = new int;
     if (n <= 1) {
        *result = *n;
       return result;
      } else {
        DOTS_Thread_Handle handleA, handleB;
       int argA=*n-1, argB=*n-2;
        dots_fork(handleA, fib_jacket, &argA);
        dots_fork(handleB, fib_jacket, &argB);
        dots_join(handleA, resultA);
        dots_join(handleB, resultB);
        *result = (*resultA) + (*resultB);
       free (resultA);
       free (resultB);
       return result;
      }
   }
```

ABBILDUNG 6.5.1. Implementierungen der rekursiven Berechnung von Fibonaccizahlen.

TABELLE 6.5.1. Anwendungen von VDS im SFB. Die Parallelisierung der unterschiedlichen Anwendungen erfordert den Einsatz verschiedener Programmierparadigmen und Lastverteilungstechniken.

Projekt	Anwendung	Paradigma	Lastverteilung
A4	Reelle Nullstellenisolierung (Kapitel 7) [**38**]	Fork-Join, DAG-Scheduling	Baumbalancierung, Pie-Mapping
A4	Berechnung reeller Eigenwerte	Farming	Scattering+adaptives Work-Stealing
C2	Maschinenbelegungs planung	gewichtete Aufgaben	Randomisiertes Dimension Exchange
C2	Simultane Losgrößen- und Maschinenbelegungsplanung [**13**]	verteiltes Farming	OPT-IT / Work-Stealing
C2	Planung von getakteten Fließlinien [**10**]	Farming	Scattering+adaptives Work-Stealing
C4	Heuristische Suche in Konfigurationssystemen [**142**]	gewichtete Aufgaben	Randomisiertes Dimension Exchange

ein Kind-Thread erzeugt worden ist, wird der erzeugende Thread parallel zum erzeugten Thread ausgeführt. Eine Cilk Funktion muss eine `sync`-Operation ausführen, um sicherzustellen, dass die Kind-Threads ihre Ergebnisse berechnet haben.

DOTS ist eine C++ Bibliothek, die speziell für Fork-Join Anwendungen konzipiert ist. Ähnlich wie die Bearbeitungsroutinen von VDS, müssen DOTS-Funktionen am Anfang der Anwendung registriert werden. An diese Funktionen kann jeweils nur ein Aufrufparameter übergeben werden. Wenn eine Funktion auf mehr als einem Parameter operiert, dann müssen diese Parameter in einer speziellen Datenstruktur zusammengefasst werden. Vom Konzept her können diese Datenstrukturen mit den Objekten von VDS verglichen werden. Andere Ähnlichkeiten zwischen VDS und DOTS sind die von beiden Bibliotheken bereitgestellten Fork- und Join-Funktionen. Die Funktion `dots_fork` liefert einen Bezeichner, der benutzt wird, um auf die von den Kind-Threads generierten Ergebnisse zuzugreifen; eine Aufgabe, die in VDS von den Variablen des Typs `VDS_Result` übernommen wird.

Für den Vergleich der vier Systeme wurde das Fibonacci-Programm mit einer Warteschleife instrumentiert, die jedesmal ausgeführt wird, wenn die Ergebnisse der beiden rekursiven Aufrufe berechnet sind. Der Fibonacci Benchmark hat somit zwei Parameter, nämlich den Index n der Fibonaccizahl und den Parameter t, der die Länge der Warteschleife angibt. Aus technischen Gründen war es nicht möglich, alle Systeme auf derselben Hardware einzusetzen. Es wurden die folgenden Systeme benutzt:

(1) Athapascan-1 (Version 1.8): Ein Linux Cluster mit 3 PCs. Zwei von ihnen hatten jeweils 2 Pentium II Prozessoren mit 333 MHz. Der dritte bestand aus 4 Pentium Pro Prozessoren mit 200 MHz. Sowohl Athapascan-1 als auch VDS benutzten die LAM Implementierung von MPI.

(2) Cilk-5 (Version 5.2): Ein SparcServer 1000 mit 8 Prozessoren (60 MHz). Der Rechner wurde mit dem Betriebssystem SunOS 5.6 betrieben.
(3) DOTS 1.2: Ein HPCLine System. Ein Workstation Cluster mit 92 Siemens Primergy Server Knoten ($2\times$Pentium II, 450 MHz). Das System wurde mit SunOS 5.7 betrieben. Die DOTS Implementierung basierte auf ACE 4.6 und VDS benutzte PVM 3.4. In den Experimenten bezieht sich der Begriff "Prozessoranzahl" in diesem Fall auf die Anzahl der Knoten. Beispielsweise wurde bei VDS nur ein Anwendungsprozess pro Knoten gestartet.

Der erste Test demonstriert die Systemkosten der Umgebungen. Für diesen Test wurde die 20-te Fibonaccizahl sequentiell ohne Wartezeit ($t = 0$) berechnet. Die gemessenen Laufzeiten sind in der folgenden Tabelle in Sekunden angegeben.

Intel-Linux, 333 MHz		Sparc-Solaris, 60 MHz		Intel-Solaris, 450 MHz	
Athapascan-1	VDS	Cilk-5	VDS	DOTS	VDS
4.8	1.1	0.56	2.38	21.5	0.78

Das einzige System, das weniger Kosten erzeugt als VDS ist Cilk-5. Der Grund hierfür ist, dass Cilk-5 eine eigene Speicherverwaltung für die Aktivierungsrahmen verwaltet [**55**]. Dieses kann sehr effizient geschehen, da alle Rahmen einer Cilk-5 Funktion dieselbe Größe haben. Die lokalen Variablen der Threads werden im Laufzeitkeller gespeichert. VDS speichert die zu einem Objekt gehörenden Daten in einem zusammenhängenden Stück virtuellen Speichers. Dieses bedingt den Aufruf der Betriebssystemroutine `malloc` für jedes neue Objekt. VDS benutzt den Laufzeitkeller aus zwei Gründen nicht. Zum einen ist VDS für Systeme ohne gemeinsamen Speicher konzipiert. Es ist somit im Allgemeinen nicht möglich, auf den Laufzeitkeller eines anderen Prozessors zuzugreifen. Zum anderen ist der Laufzeitkeller für andere Paradigmen nicht die geeignete Datenstruktur.

Die Schlussfolgerung aus diesem Test ist, dass die Granularität der Aufgaben, die von Athapascan-1, DOTS und VDS generiert werden, von der Anwendung kontrolliert werden sollte. Die Aufgaben sollten bei den drei Systemen nicht zu feingranular sein.

Der zweite Test bezieht sich auf die Skalierbarkeit der Systeme. Um die Granularität des Fibonacci-Benchmarks zu erhöhen, wurde die Wartezeit auf 0.1 Sekunden gesetzt.

Abbildung 6.5.2 zeigt die Effizienzen, die von den Systemen erreicht werden. Da das Cilk-5 System Work-Stealing direkt über den gemeinsamen Speicher realisiert, kann es höhere Effizienzen erreichen als VDS. In Cilk-5 arbeiten sowohl der bestohlene als auch der stehlende Prozessor mit den Daten im Laufzeitkeller des bestohlenen Prozessors [**55**]. Dagegen realisiert VDS Work-Stealing durch den Austausch von Nachrichten. Dieses erfordert selbst auf einem System mit gemeinsamen Speicher zwangsläufig mehr Kosten. Verglichen mit Athapascan-1 und DOTS ist VDS effizienter. Ein Grund hierfür sind die Kosten der späten Laufzeitbindung in C++. Ein anderer Grund ist die Tatsache, dass Athapascan-1 für das viel allgemeinere Paradigma der Datenflusssteuerung konzipiert ist und deswegen zur Laufzeit den Aufgabengraph der Anwendung explizit konstruiert.

Insgesamt zeigen die Experimente, dass VDS von den getesteten Systemen die effiziente Implementierung strikter Baumberechnungen auf Systemen mit verteiltem Speicher bietet.

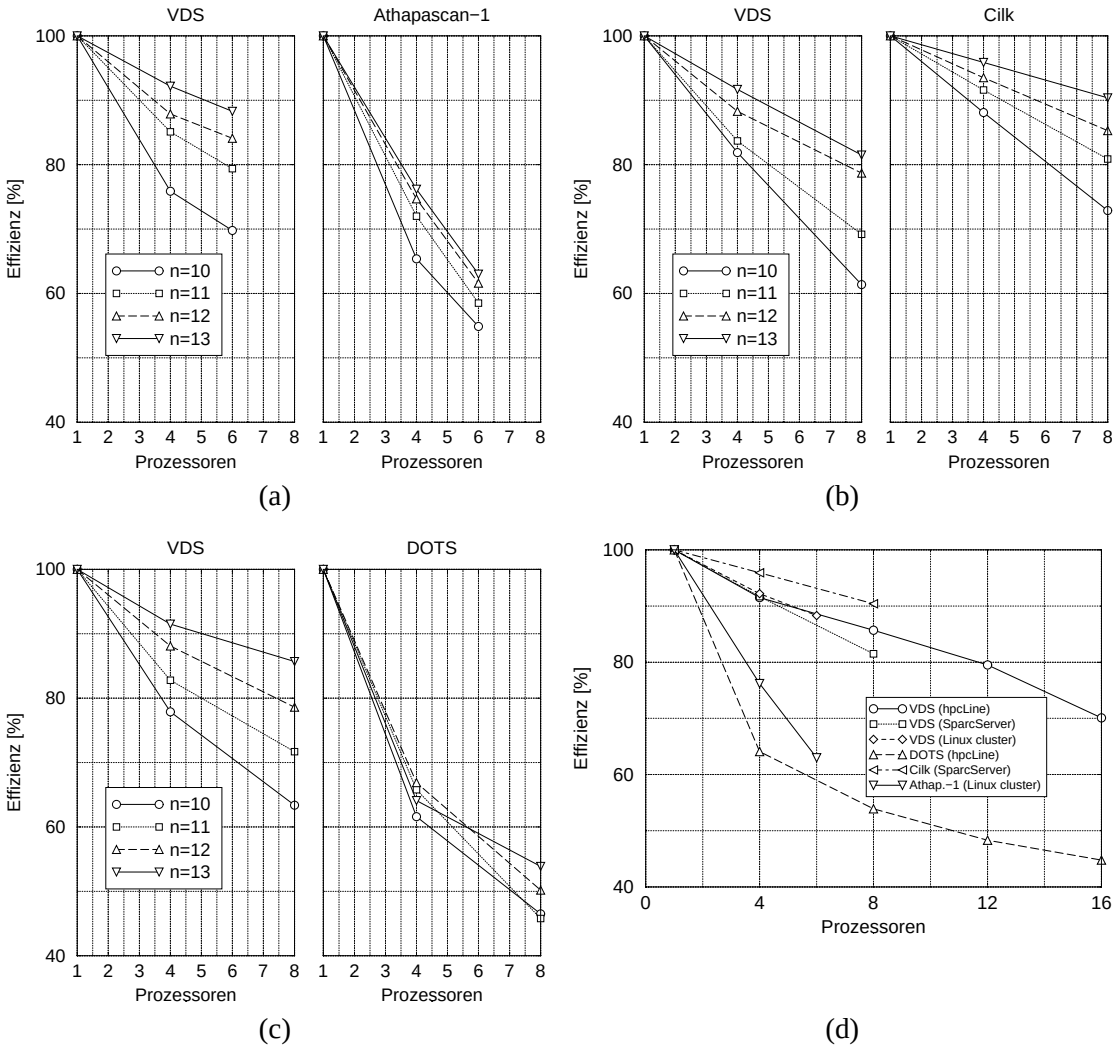


ABBILDUNG 6.5.2. Vergleich der Lastverteilungssysteme Athapascan-1, DOTS, Cilk-5 und VDS. Die Diagramme (a), (b), und (c) zeigen die Effizienzen, die für unterschiedliche Werte von n und einer Wartezeit von $t = 0.1$ Sekunden gemessen wurden. Das Diagramm (d) zeigt die Effizienzen für $n = 13$ und $t = 0.1$ Sekunden.

KAPITEL 7

Parallele Nullstellenisolierung mit der Descartes Methode

Dieses Kapitel beschreibt eine Anwendung, die mit Hilfe der VDS Bibliothek parallelisiert worden ist. Sie demonstriert eine der besonderen Eigenschaften von VDS: die Kombination unterschiedlicher Programmierparadigmen. Darüberhinaus demonstriert sie die Nutzung des Isoeffizienzkonzepts für die Analyse von Algorithmen, deren parallele Effizienz nicht unmittelbar von der sequentiellen Laufzeit abhängt.

Die Aufgabe der *reellen Nullstellenisolierung* ist es, disjunkte Intervalle für alle reellen Nullstellen eines Polynoms zu finden. Sobald diese Intervalle berechnet sind, können die Nullstellen mit beliebiger Genauigkeit durch numerische Methoden bestimmen werden. Auf diese Weise kann sichergestellt werden, dass alle Nullstellen gefunden werden—eine Garantie, die bei rein numerischen Methoden nicht gegeben werden kann. Die Berechnung reeller Nullstellen hat vielfältige Anwendungen unter anderem bei der Steuerung von Robotern [**104**], beim Rechnen mit algebraischen Zahlen oder bei der Berechnung komplexer Nullstellen [**30**].

Es sind mehrere sequentielle Algorithmen für die Berechnung reeller Nullstellen bekannt. Die Vorzeichenmethode nach Collins und Akritas [**27, 85**] hat sich für praktische Zwecke als sehr geeignet erwiesen [**78, 79**]. Da diese Methode auf der Vorzeichenregel von Descartes beruht, nennen wir sie die *Descartes Methode.* Implementierungen dieser Methode finden sich zum Beispiel in dem Computeralgebrasystem *Maple* (Funktion `realroot`) [**16**], *MuPAD* (Funktion `realroots`), und in der Bibliothek SACLIB (Funktion `IPRRID`) [**29**].

Wir betrachten Polynome mit ganzzahligen Koeffizienten. Polynome mit rationalen oder Fließkommakoeffizienten können durch Multiplikation mit einem geeigneten Faktor leicht in Polynome mit ganzzahligen Koeffizienten umgewandelt werden. Ferner setzt die Descartes Methode voraus, dass alle Nullstellen des Eingabepolynoms einfach sind. Falls ein Eingabepolynom $A(x)$ diese Bedingung nicht erfüllt, betrachten wir den größten quadratfreien Teiler $B(x) = A(x)/\text{ggT}(A(x), A'(x))$, wobei $A'(x)$ die erste Ableitung von $A(x)$ ist.

Die Descartes Methode führt eine binäre Suche durch. Sie startet mit einem Intervall, das alle Nullstellen des Eingabepolynoms enthält. Dieses wird durch rekursive Bisektionierung solange verfeinert, bis alle Intervalle entweder keine Nullstelle oder genau eine Nullstelle enthalten. Das initiale Intervall wird durch die Berechnung einer Nullstellenschranke bestimmt [**84**, Übung 4.6.2.20].

Im folgenden Abschnitt gehen wir auf zwei Methoden, die Descartes Methode zu parallelisieren, ein. In Abschnitt 7.2 wird eine skalierbare Parallelisierung vorgestellt, die die in den Kapiteln 3 und 4 vorgestellten Planungsverfahren nutzt. In Abschnitt 7.3 wird gezeigt, dass die Isoeffizienz der parallelen Descartes Methode bei beschränkter Koeffizientenlänge von $P^3 \left(\log P\right)^2$ dominiert wird.

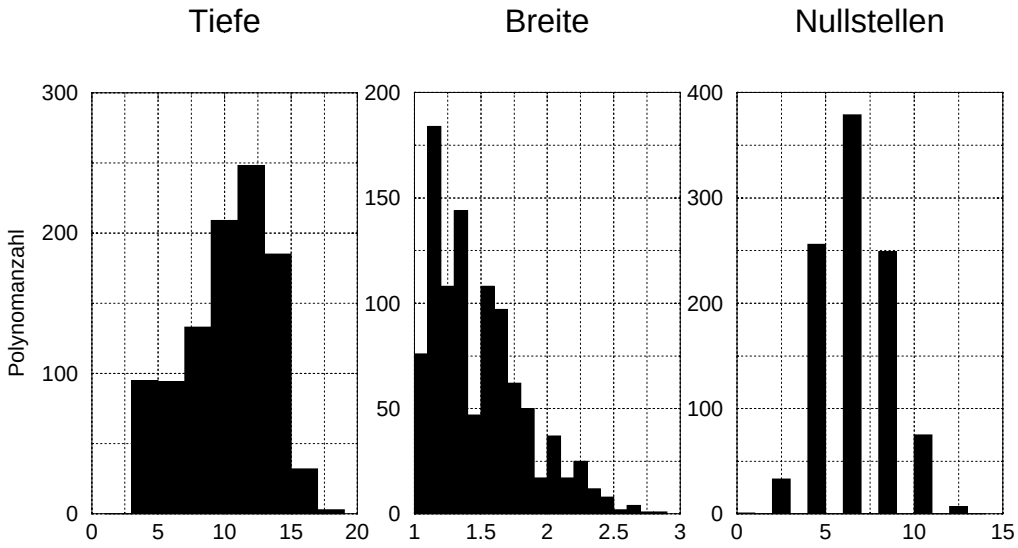


ABBILDUNG 7.1.1. Die Verteilung der Suchbaumtiefe und -breite sowie der Anzahl der Nullstellen von 1000 Polynomen vom Grad 8000 mit zufällig gewählten Koeffizienten. Die geringe Breite erschwert die Parallelisierung der Methode.

7.1. Stand der Forschung

Es wurden bislang zwei Verfahren vorgestellt, die Descartes Methode zu parallelisieren. Collins, Johnson und Küchlin untersuchen eine Parallelisierung, bei der ausschließlich die Baumparallelität ausgenutzt wird [**28**]. Sie benutzen hierfür das für das Fork-Join Paradigma ausgelegte System DTS. Allerdings sind die Suchbäume oft sehr schmal. Beispielsweise zeigt die Isolierung von 1000 Polynomen mit zufällig gewählten Koeffizienten vom Grad 8000, dass die Suchbäume typischerweise nicht breiter als zwei sind (vgl. Abbildung 7.1.1). Somit ist die auf der reinen Baumparallelität basierende Parallelisierung nicht skalierbar.

Schreiner, Mittermeier und Winkler schlagen vor, die Effizienz zu erhöhen, indem die Intervalle, die mehr als eine Nullstelle enthalten, in so viele Unterintervalle aufgeteilt werden, wie Prozessoren vorhanden sind [**100, 118**]. Auf diese Weise kann bei der Auswertung einer Ebene im Suchbaum jedem Prozessor mindestens ein Suchknoten zugewiesen werden. Dieser Ansatz wertet jedoch, falls das Polynom nur wenige reelle Nullstellen besitzt, mehr Suchknoten aus, als es die sequentielle Methode tun würde. Da dieses zu einer geringen Effizienz führt, ist folglich auch dieser Ansatz im Allgemeinen nicht skalierbar.

Das im nächsten Abschnitt diskutierte Vorgehen ist durch die Ausnutzung der Parallelität in den Suchknoten und der Baumparallelität die erste skalierbare Parallelisierung der Descartes Methode.

7.2. Die parallele Descartes Methode

Eine detaillierte Beschreibung der sequentiellen Descartes Methode findet sich in [**85**]. Die Darstellung an dieser Stelle beschränkt sich auf die für die Parallelisierung wichtigen Aspekte.

Wir bestimmen die reellen Nullstellen eines Polynoms $A(x)$, indem wir parallel isolierende Intervalle für positiven Nullstellen von $A(x)$ und von $B(x) = A(-x)$ bestimmen. Beide Isolierungsvorgänge starten mit einem Intervall, das jeweils alle positiven Nullstellen enthält. Dieses

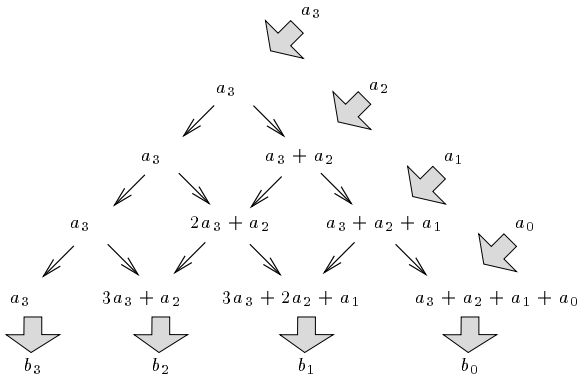


ABBILDUNG 7.2.1. Der Taylor-Shift mit synthetischen Divisionen. Falls $A(x) = a_3x^3 + a_2x^2 + a_1x + a_0$ ist, benötigen wir 6 Additionen um die Koeffizienten von $B(x) = A(x + 1)$ zu berechnen.

wird zunächst darauf überprüft, ob es leer ist, oder ob es bereits ein isolierendes Intervall ist. Andernfalls handelt es sich um einen inneren *Suchknoten* im Suchbaum.

Die Auswertung eines Suchknotens besteht darin, das assoziierte Intervall zu bisektionieren und somit zwei neue (potentielle) Suchknoten zu erzeugen. Diese werden wiederum dahingehend überprüft, ob sie innere Knoten sind, oder ob die assoziierten Intervalle leer oder isolierend sind. Die parallele Methode bearbeitet die Knoten im Suchbaum ebenenweise. Dabei wird die Kenntnis der Anzahl der Suchknoten auf derselben Baumebene für die Bearbeitungsplanung genutzt.

Jeder Suchknoten enthält ein Intervall $(a,\ b)$ und ein Polynom $B(x)$, das im Intervall $(0,\ 1)$ genauso viele Nullstellen besitzt, wie das Eingabepolynom im Intervall $(a,\ b)$. Die Koeffizienten von $B(x)$ werden durch Fließkommaintervalle mit einer festen Mantissenlänge (Präzision) repräsentiert. Um die Teilintervalle $(a,\ (a+b)/2)$ und $((a+b)/2,\ b)$ zu untersuchen, müssen die zugehörigen Polynome $L(x)$ für die linke Hälfte und $R(x)$ für die rechte Hälfte berechnet werden. Anschließend werden beide Polynome durch Verwendung der Vorzeichenregel von Descartes auf Nullstellen im Intervall $(0,\ 1)$ überprüft. Hierbei kann es vorkommen, dass die Vorzeichenbestimmung einiger Koeffizienten nicht möglich ist, wenn die Fließkommaintervalle die 0 enthalten. In diesem Fall wird die gesamte Rechnung mit der doppelten Präzision wiederholt.

Sowohl für die Konstruktion von $R(x)$ als auch für die Überprüfung der Intervalle ist eine *Verschiebung um 1* notwendig (*Taylor-Shift*). Der Taylor-Shift eines Polynoms $A(x)$ berechnet das Polynom $B(x)$ mit $B(x) = A(x + 1)$. Um das Polynom $B(x)$ zu berechnen, benutzen wir synthetische Divisionen. Die Idee ist hierbei, das Polynom $A(x)$ als ein Polynom in $x - 1$ auszudrücken, also $A(x) = b_n(x-1)^n + \cdots + b_1(x-1) + b_0$. Die hier auftretenden Koeffizienten $b_0, \ldots, b_n$ sind gerade die gesuchten Koeffizienten des Polynoms $B(x)$. Zunächst erhalten wir den Koeffizienten b_0 von $B(x)$, der mit dem Koeffizienten a_0 von $A(x)$ identisch ist. Wenn wir nun $A(x) - b_0$ durch $(x - 1)$ teilen, erhalten wir den Koeffizienten b_1 von $B(x)$. Durch Iteration dieses Vorgangs erhalten wir schließlich alle Koeffizienten von $B(x)$. Für diese Rechnung sind ausschließlich die in Abbildung 7.2.1 dargestellten $n \cdot (n + 1)/2$ Additionen notwendig.

Bei der Parallelisierung treten somit die drei in Abbildung 7.2.2 dargestellten Parallelitätsebenen auf. Auf der obersten Ebene haben wir die Baumparallelität, die durch die gleichzeitige Auswertung aller Suchknoten, die sich in derselben Baumebene befinden, entsteht. Die zweite

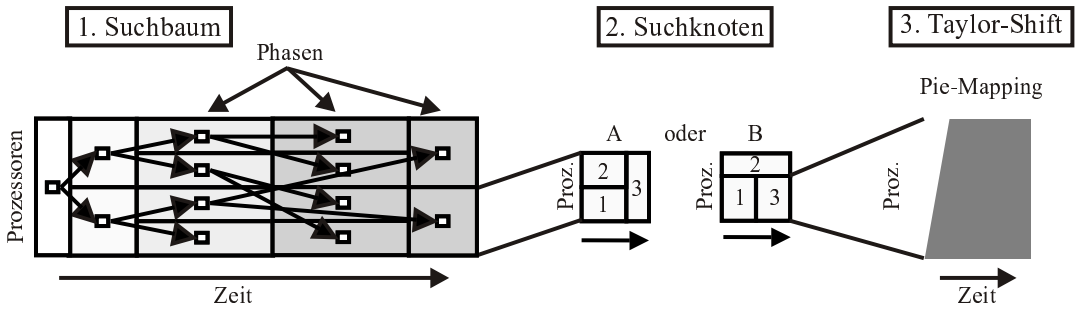


ABBILDUNG 7.2.2. Die drei Parallelitätsebenen der parallelen Descartes Methode.

Ebene besteht aus der parallelen Ausführung der für einen Suchknoten erforderlichen Taylor-Shifts, und die dritte Ebene aus der parallelen Berechnung eines einzelnen Taylor-Shifts. Im Folgenden gehen wir näher auf diese drei Ebenen ein.

7.2.1. Planung des Taylor-Shifts. Wie aus Abbildung 7.2.1 hervorgeht, erfordert die Berechnung des Taylor-Shifts eines Polynoms vom Grad $n-1$ mit synthetischen Divisionen die Auswertung einer Pyramide der Höhe n. Wegen der Verwendung von Fließkomma-Arithmetik kann die Dauer jeder einzelnen Addition näherungsweise als gleich angesehen werden. Wir können daher das in Abschnitt 3.3.5 beschriebene Pie-Mapping Verfahren mit Ebenen- oder Blockplanung benutzen. Durch Nutzung der in Abschnitt 3.3 beschriebenen Kachelungstechnik erzeugen wir eine Pyramide der Höhe

$$h = \begin{cases} s\,P & \text{falls} \quad \frac{n}{sP} \geq m \\ \lfloor \frac{n}{m} \rfloor & \text{falls} \quad \frac{n}{sP} < m \end{cases}$$

wobei m die minimale Kantenlänge einer Kachel und s die Anzahl der Knoten innerhalb einer Zone in der letzten Ebene der Pyramide ist. Die minimale Kantenlänge ist ein architekturabhängiger Parameter. Mit Hilfe von s kann die Granularität der Pyramidenberechnug gesteuert werden. Die Standardeinstellung für den hpcLine-Cluster und die Cray T3E ist $m = 16$ und $s = 3$. Bei großen Präzisionen sollte der Wert von m verringert werden und und bei großen Polynomgraden der Wert von s auf 5 oder 7 erhöht werden, um die beste Effizienz zu erreichen. Alle Experimente wurden mit den Standardwerten dieser Parameter unter Verwendung der Blockplanung durchgeführt.

Für die Analyse der Descartes Methode ignorieren wir die minimale Kantenlänge und gehen von einer Pyramidenhöhe $h = s\,P$ für ein festes, ungerades $s \in \mathbb{N}$ aus. Wir bezeichnen mit t die Dauer einer Addition und modellieren die Latenzzeit der Übertragung einer Botschaft der Größe l durch $l\,L$. Weiterhin betrachten wir wegen der leichteren Analysierbarkeit die Ebenenplanung. Wie sich herausstellen wird, ist sie bereits bezüglich der Größenordnung der Isoeffizienz optimal.

Genaugenommen entstehen durch die Kachelung Pyramidenknoten, die sich in ihrem Rechenaufwand unterscheiden, falls n nicht durch h teilbar ist. Da sich die Kantenlängen der Kacheln jedoch höchstens um einen Koeffizienten unterscheiden, vernachlässigen wir diesen Effekt im Folgenden und modellieren die Knoten in den ersten $n-1$ Ebenen als gleichförmige Aufgaben mit einem Rechenaufwand von $(n/(s\,P))^2\,t$. Die Aufgaben in der letzten Ebene haben den

Rechenaufwand

$$\left(\left(\frac{n}{s\,P}\right)^2 - \frac{1}{2}\frac{n}{s\,P}\left(\frac{n}{s\,P} - 1\right)\right)\ t\ . \tag{7.2.1}$$

Insgesamt erhalten wir bezüglich dieser Modellierung die folgende parallele Laufzeit für den Taylor-Shift.

THEOREM 7.2.1. *Es sei s eine ungerade ganze Zahl. Wenn der Taylor-Shift eines Polynoms vom Grad $n-1$ mit Fließkommaarithmetik unter Benutzung von Pie-Mapping mit Ebenenplanung auf $P \geq 2$ Prozessoren durchgeführt wird, dann ist die parallele Ausführungszeit gegeben durch*

$$T(n,\,P) = T_{calc}(n,\,P) + T_{com}(n,\,P)\ ,$$

wobei T_{com} der Kommunikationsoverhead und T_{calc} die Rechenzeit ist. Es gilt

$$\underbrace{(s\,P-1)\,o + (P-1)\ \frac{n}{sP}L}_{=:\underline{T}_{com}} \leq T_{com}(n,\,P) \leq \underbrace{(2\,s\,P+P-4)\,o + (sP-1)\ \frac{n}{sP}\,L}_{=:\overline{T}_{com}}\ , \tag{7.2.2}$$

$$\text{und } T_{calc}(n,\,P) = \frac{1}{2}\left(\left(\left(s^2+1\right)\ P + s - 1\right)\left(\frac{n}{s\,P}\right)^2 - \frac{n}{P}\left(\frac{n}{s\,P} - 1\right)\right)\ t\ . \tag{7.2.3}$$

Die sequentielle Rechenzeit ist gegeben durch $T(n,\,1) = \frac{n\,(n-1)}{2}t$.

BEWEIS. Um die Berechnungszeit des Taylor-Shifts herzuleiten, vergleichen wir sie mit einer Pyramidenberechnung mit der Knotenberechnungszeit $t' = (n/(s\,P))\,t$ und der Latenzzeit $L' = (n/(s\,P))\,L$. Für diese Berechnung erhalten wir mit Theorem 3.3.14 die parallele Rechenzeit

$$T(n,\,P) = \frac{1}{2}\left(\left(s^2+1\right)\ P + s - 1\right)\ \left(\frac{n}{s\,P}\right)^2 t + T_{\text{com}}(n,\,P)\ ,$$

wobei T_{com} die Gleichung (7.2.2) erfüllt. Da alle Knoten in den ersten $n-1$ Ebenen der Taylor-Shift Berechnung ebenfalls die Rechenzeit t' haben, beginnen die Prozessoren in beiden Rechnungen den ersten Knoten in der letzten Ebene zur gleichen Zeit. Da s ungerade ist, werden für die Bearbeitung der restlichen Knoten in der letzten Ebene keine Daten von anderen Prozessoren benötigt. Sie können also ohne Wartezeiten berechnet werden. Da bei der Berechnung des Taylor-Shifts die Berechnungsdauer jedes einzelnen der s Knoten in der letzten Ebene einer Zone durch Gleichung (7.2.1) gegeben ist, wird der Taylor Shift um die Zeit

$$s\,\frac{1}{2}\frac{n}{s\,P}\left(\frac{n}{s\,P} - 1\right)\ t$$

schneller berechnet, als eine Pyramidenrechnung, bei der jeder Knoten die Berechnungszeit t' hat. Insgesamt folgt damit die Beziehung (7.2.3) für T_{calc}. □

Offensichtlich sind die sequentiellen Rechenzeiten im Sinne von Definition 2.2.4 diskret. Mit Hilfe der Schranken für den Kommunikationsaufwand definieren wir Schranken für Berechnungszeit und für die Effizienz des Taylor-Shifts.

DEFINITION 7.2.2. Für alle $n \geq 1$ und $P \geq 2$ sei

$$\underline{T}(n,\,P) := T_{\text{calc}}(n,\,P) + \underline{T}_{\text{com}}(n,\,P) \quad ,\quad \overline{E}(n,\,P) := \frac{T(n,1)}{P\,\underline{T}(n,\,P)}\ ,$$
$$\overline{T}(n,\,P) := T_{\text{calc}}(n,\,P) + \overline{T}_{\text{com}}(n,\,P) \quad ,\quad \underline{E}(n,\,P) := \frac{T(n,1)}{P\,\overline{T}(n,\,P)}\ .$$

Die Isoeffizienzfunktionen $\underline{I}_e$ und $\overline{I}_e$ seien gemäß Theorem 2.2.3 definiert.

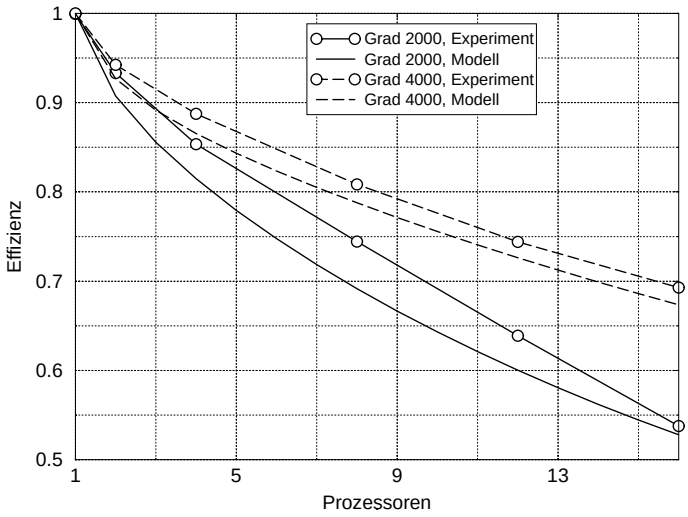


ABBILDUNG 7.2.3. Ein Vergleich von der LogP-Modellierung des Taylor-Shifts mit realen Messungen. Die Experimente wurden mit einer Mantissenlänge von zwei Wörtern durchgeführt. In Bezug auf die verwendete Implementierung wurden die folgenden LogP-Parameter für den Parsytec CC Rechner bestimmt: $L = 5\mu s/\text{Byte}$, $o = g = 0.3\,\text{ms}$.

Es gilt somit für alle $n,\ P \in \mathbb{N}$ und alle $e \in (0, 1)$,

$$\begin{array}{rcccl} \underline{T}(n,\ P) & \leq & T(n,\ P) & \leq & \overline{T}(n,\ P), \\ \underline{E}(n,\ P) & \leq & E(n,\ P) & \leq & \overline{E}(n,\ P), \\ \underline{I}_e(n,\ P) & \leq & I_e(n,\ P) & \leq & \overline{I}_e(n,\ P). \end{array}$$

Ferner erfüllen $\underline{E}$ und $\overline{E}$ die speed-up und size-up Eigenschaft (Definitionen 2.2.6 und 2.2.8). Der Vergleich der unteren Schranke $\underline{E}(n,\ P)$ im LogP-Modell und der gemessenen Effizienz eines Taylor-Shifts in Abbildung 7.2.3 zeigt, dass die Aussagen des Modells realistisch sind.

7.2.2. Planung der Knotenberechnungen. Jeder Suchknoten erfordert die Berechnung von zwei oder drei Taylor-Shifts. Wir bezeichnen den Taylor-Shift für den Test des linken Intervalls mit $\mathcal{L}$ und die beiden für das rechte Intervall erforderlichen Verschiebungen mit $\mathcal{R}_1$ und $\mathcal{R}_2$. Der Shift $\mathcal{R}_2$ kann unter bestimmten Voraussetzungen entfallen.

Angenommen, es stehen P Prozessoren für die Auswertung des Suchknotens zur Verfügung. Dann bieten sich zwei Möglichkeiten für die Planung der Operationen an. Die Methode A berechnet zunächst $\mathcal{L}$ und $\mathcal{R}_1$ gleichzeitig auf jeweils $\lfloor P/2 \rfloor$ Prozessoren und dann $\mathcal{R}_2$ auf P Prozessoren. Die Methode B benutzt $\lfloor P/3 + 1/2 \rfloor$ Prozessoren für $\mathcal{L}$ und führt die Verschiebungen $\mathcal{R}_1$ und $\mathcal{R}_2$ nacheinander auf den restlichen $P - \lfloor P/3 + 1/2 \rfloor$ Prozessoren aus. Mit Hilfe der unteren Schranke für die Effizienz des Taylor-Shifts lässt sich abschätzen, welche der Methoden für eine gegebene Prozessoranzahl am günstigsten ist—vorausgesetzt, dass wir wissen ob $\mathcal{R}_2$ ausgeführt oder nicht ausgeführt wird.

Der Nachteil von Methode B ist, dass sie ineffizient ist, wenn $\mathcal{R}_2$ nicht berechnet wird. Da dieses bekannterweise am Anfang des Suchbaumes der Fall ist [**85**], benutzen wir am Anfang ausschließlich Methode A, also für die Wurzel und immer dann, wenn der Baum die Breite 1

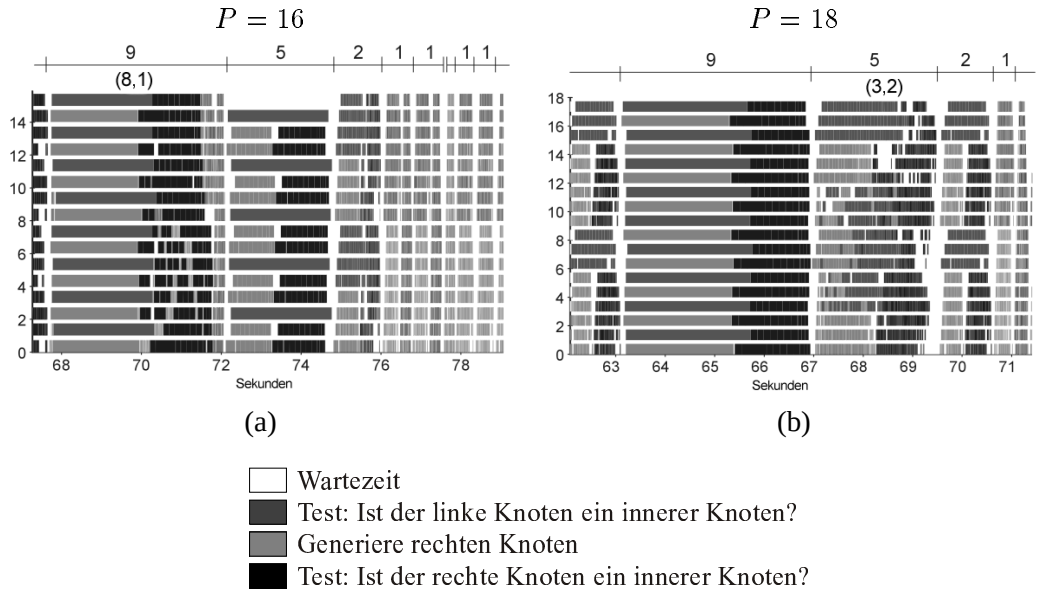


ABBILDUNG 7.2.4. Ausführungsprofile von Nullstellenisolationen eines Tschebyscheff Polynoms vom Grad 300 auf 16 (a) und 18 Prozessoren (b). Die Berechnungen wurden auf dem Parsytec-GC/PP Rechner durchgeführt.

hat und der vorhergegangene Suchknoten nur aus zwei Verschiebungen bestand. Sonst gehen wir davon aus, dass jeder Suchknoten aus drei Verschiebungen besteht und wählen jeweils die effizientere der beiden Methoden.

7.2.3. Planung des Suchbaumes. Da jeder Suchknoten die gleiche Laufzeit bei einer gegebenen Prozessoranzahl hat und diese gut abgeschätzt werden kann, kann die Planung des Suchbaumes mit der in Abschnitt 4.4 beschriebenen Methode für die Planung von Baumberechnungen mit parallelisierbaren Knoten erfolgen.

Die Abbildung 7.2.4 zeigt Ausschnitte zweier Ausführungsprofile der Descartes Methode. Die verschiedenen Grautöne repräsentieren jeweils Taylor-Shifts (dunkelgrau für $\mathcal{L}$, hellgrau für $\mathcal{R}_1$ und schwarz für $\mathcal{R}_2$). Die Skala über den Profilen markiert die Baumebenen, die Anzahl der Suchknoten und die verwendeten Pläne.

Das Profil (a) zeigt die Schlussphase einer auf 16 Prozessoren durchgeführten Isolierung. Die erste Ebene enthält 9 Knoten, die mit dem phasenparallelen Plan $(8,\ 1)$ berechnet werden. Für die nächste Ebene gibt es zwei Kandidaten für gute Pläne: den Plan (5), bei dem ein Prozessor unbenutzt bleibt und den Plan $(4, 1)$. In diesem Fall hat sich der Planungsalgorithmus offensichtlich für die erste Variante entschieden. Das Profil (b) zeigt denselben Teil der Berechnung auf 18 Prozessoren. Erwartungsgemäß ergeben sich hier wegen der geänderten Prozessoranzahl andere Pläne.

7.2.4. Experimentelle Ergebnisse. Für die experimentelle Evaluierung betrachten wir vier Polynomklassen. Abbildung 7.2.5 zeigt typische Suchbäume dieser Klassen.

Zufallspolynome sind Polynome, deren Koeffizienten gleichverteilt aus einem gegebenen Intervall bestimmt werden. Von derartigen Polynomen ist bekannt, dass sich ihre durchschnittliche

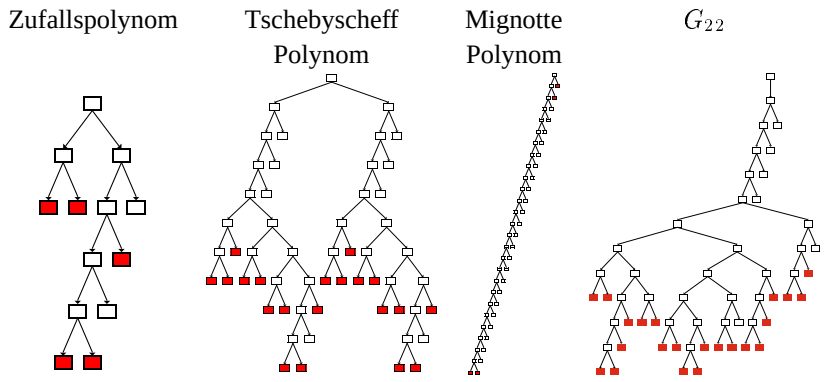


ABBILDUNG 7.2.5. Suchbäume, die bei der Isolierung reeller Nullstellen von Polynomen entstehen. Die ersten drei Polynome haben Grad 20 und das Polynom G_{22} ist vom Grad 22. Die grauen Knoten entsprechen isolierenden Intervallen.

Nullstellenanzahl asymptotisch der Funktion $2 \ln n/\pi$ nähert [**49**]. Die Suchbäume von Zufallspolynomen sind typischerweise schmal. Die im Folgenden beschriebenen Experimente beziehen sich auf Polynome mit 2000-Bit Koeffizienten. Die Messwerte repräsentieren den Durchschnitt von jeweils 50 Messungen mit festem Grad.

Tschebyscheff Polynome haben nur reelle Nullstellen. Tschebyscheff Polynome (der ersten Art) werden wie folgt rekursiv definiert. $T_0(x) = 1, T_1(x) = x, T_{n+1}(x) = 2\,x\,T_n(x) - T_{n-1}(x)$. Die Nullstellen von $T_n(x)$ sind gegeben durch $\cos((2k+1)\pi/(2n))$ wobei $k = 0, \ldots, n-1$. Die Suchbäume von Tschebyscheff Polynomen sind breit und tief.

Mignotte Polynome haben die Form $A_n(x) = x^n - 2\,(5\,x-1)^2$. Sie haben 4 Nullstellen, von denen zwei links und rechts von $1/5$ liegen, mit einem Abstand von $2/5^{n/2+1} < 1/2^{n+1}$ [**85**]. Die Suchbäume von Mignotte Polynomen sind schmal und extrem tief.

G_n-Polynome sind charakteristische Polynome eines $n \times n$ Ausschnitts einer unendlich großen Matrix G_∞. Die Eigenwerte der Matrix G_∞ beschreiben Energieniveaus von Elektronen in Atomen mit zwei Elektronen. Sie hat zwei Variablen e und Z, wobei Z die Anzahl der Protonen im Atomkern bezeichnet. Wir betrachten hier den Fall $Z = 2$ (Helium). Physikalisch sinnvolle Aussagen liefern die Polynome G_7, G_{13}, G_{22}, G_{34}, G_{50}, G_{70}, G_{95}, $G_{125}, \ldots$ [**4**]. Sie haben extrem lange Koeffizienten, nur reelle Nullstellen und breite Suchbäume. Die Polynome (und deren Nullstellen) für beliebige Werte von Z wurden von Barnett für die Untersuchung von spektralen Regelmäßigkeiten von Atomen mit zwei Elektonen genutzt [**4**].

Arithmetische Aspekte. Durch die Nutzung von Intervallarithmetik wird die Methode nie ein falsches Ergebnis berechnen. Schlimmstenfalls scheitert sie wegen einer zu klein gewählten Präzision. In diesem Fall meldet sie einen Fehler. Um insgesamt einen Algorithmus zu erhalten, der immer das gewünschte Ergebnis liefert, starten wir mit einer Mantissenlänge von zwei Wörtern und verdoppeln die Präzision nach jedem fehlgeschlagenen Lauf. Dieses Verfahren wird spätestens dann terminieren, wenn die Mantisse so lang ist, dass alle Zwischenergebnisse exakt dargestellt werden können. Wir nennen dieses Vorgehen dementsprechend *Präzisionsverdopplung*. Weiterhin bezeichnen wir die kleinste Präzision, mit der die Nullstellen eines gegebenen

TABELLE 7.2.1. Präzisionen und sequentielle Berechnungszeiten. Die Präzisionen bezeichnen die Anzahl von 29-Bit Worten. Die Berechnungszeiten sind in Sekunden angegeben und wurden auf einem PC mit einem 300 MHz Pentium Pro Prozessor gemessen.

Klasse / Grad	Präzisions-verdopplung		optimale Präzision	
	Zeit	Präz.	Zeit	Präz.
Tschebyscheff				
100	5.1	4	5.0	3
200	96.0	16	61.6	9
300	321.0	16	267.0	13
400	1310.0	32	778.0	17
500	2557.0	32	1861.0	22
Mignotte				
100	23.9	16	14.3	9
200	326.2	32	161.0	17
300	932.0	32	724.0	25
Zufallspolynom				
100	0.22	2	0.22	2
200	0.99	2	0.99	2
300	2.15	2	2.15	2
400	3.69	2	3.69	2
1000	35.60	2	35.60	2
G_n				
34	0.25	4	0.18	4
50	1.18	8	0.57	5
70	2.5	8	1.73	7
95	11.81	16	4.78	9
125	23.52	16	12.68	12

Polynoms berechnet werden können, als die *optimale Präzision.* Die Präzisionsverdopplung übertrifft die optimale Präzision höchstens um den Faktor zwei. Da die Methode nur Additionen benutzt, ist ihr Aufwand linear in der Präzision. Folglich ist Präzisionsverdopplung höchstens vier mal langsamer als ein einziger Lauf mit der optimalen Präzision.

Die Tabelle 7.2.1 zeigt einen Vergleich sequentieller Berechnungszeiten mit Präzisionsverdopplung und Startpräzision 2 mit den Zeiten, die benötigt werden, wenn die optimale Präzision benutzt wird. Ferner sind die benutzten Präzisionen in Worten angegeben, wobei jedes Wort 29 Bit zur Mantisse beiträgt. In den Experimenten nimmt die Präzisionsverdopplung in den meisten Fällen höchstens doppelt so viel Zeit in Anspruch, wie ein einziger Lauf mit der optimalen Präzision. Dieser Effekt entsteht dadurch, dass der Algorithmus oft sehr schnell merkt, dass eine gegebene Präzision nicht ausreicht. In diesem Fall wird die Rechnung sofort gestoppt und mit der doppelten Präzision neu begonnen.

An dieser Stelle bieten sich jedoch noch Verbesserungsmöglichkeiten an. Und zwar kann die Information über das Aussehen des bereits berechneten Teilbaumes bei der erneuten Berechnung des Baumes genutzt werden. Auf diese Weise können die bereits erfolgreich durchgeführten Tests bei dem neuen Lauf entfallen. Eine weitere Verbesserung besteht darin, die erneute Suche nicht von der Wurzel aus zu beginnen, sondern direkt bei den kritischen Teilbäumen anzufangen. Dieses erfordert allerdings Polynomtransformationen, die Multiplikationen erfordern.

Die Tabelle 7.2.2 zeigt jedoch, dass bereits ohne diese Optimierungen die vorliegende Implementierung der sequentiellen Descartes Methode anderen Systemen in den meisten Fällen überlegen ist. Sie zeigt die Berechnungszeiten, die von den Systemen Maple 5.1 (Funktion `fsolve()`), MuPAD 1.4.2 (Funktion `float(hold(solve)())`) und MatLAB 5.3 (Funktion `roots()`) benötigt werden. Im Falle der Descartes-Methode ist neben der Gesamtberechnungszeit auch die Zeit angegeben, die für die Isolierung benötigt wird. An den Stellen, wo keine Zeiten angegeben sind, konnten nicht alle Nullstellen innerhalb einer dreitägigen Frist berechnet werden. Dieses äußerte sich bei Maple durch Überschreiten der drei Tage, bei MuPAD durch eine Terminierung der Rechnung, und bei MatLAB durch ein Ergebnis, dass nur einen Teil der reellen Nullstellen enthielt. Die Laufzeiten von MatLAB sind meistens deutlich kleiner als die der Descartes Methode, allerdings ist es mit MatLab im Gegensatz zu den anderen drei Systemen nicht möglich, die Nullstellen mit einer vorgegebenen Genauigkeit zu berechnen.

Außer bei den Mignotte-Polynomen wird bei der Verwendung der Descartes Methode ein Großteil der Rechenzeit für die Verfeinerung der isolierenden Intervalle benötigt. Dieses liegt teilweise daran, dass für die Verfeinerung in der derzeitigen Programmversion ein einfaches Bisektionierungsverfahren benutzt wird. Es ist eine deutliche Beschleunigung der Verfeinerung zu erwarten, wenn bei hinreichend schmalen Intervallen Newton-Iterationen durchgeführt werden.

Parallele Aspekte.

Die Tabelle 7.2.3 verdeutlicht das Ausmaß, mit dem die in den Abschnitten 7.2.2 und 7.2.3 beschriebenen Planungsalgorithmen zur Laufzeit beitragen. Die Testeingaben sind Zufallspolynome vom Grad 1000 und 2000 sowie Tschebyscheff und Mignotte Polynome vom Grad 100 und 200.

Wir vergleichen fünf Strategien für die Planung des Suchbaumes (vgl. Abschnitt 6.3.2).

(1) *Baumplanung*
(2) *Work-Stealing*
(3) *Scattering*
(4) *Workstealing gekoppelt mit Scattering*
(5) *keine Lastverteilung:* Jeder Suchknoten wird auf Knoten 0 bearbeitet. Die einzige Parallelität stammt also von den Taylor-Shifts.

Nur im ersten Fall ist bekannt, wieviele Suchknoten parallel ausgeführt werden können. Dieses ist also der einzige Fall, in dem die Prozessoren exklusiv den Suchknoten zugeordnet werden können. In den anderen Fällen wird jeder Suchknoten auf allen Prozessoren bearbeitet. Zusätzlich vergleichen wir für jede Baumplanungsmethode die Zeiten, die wir erhalten, wenn wir die in Abschnitt 7.2.2 beschriebene Knotenplanung anwenden bzw. nicht anwenden. Im letzteren Fall werden die Verschiebungen auf allen dem Suchknoten zugeteilten Prozessoren ausgeführt.

Falls der Suchbaum schmal ist, können wir keinen großen Gewinn durch die Verwendung der Baumplanung erwarten. Dieses trifft bei Zufallspolynomen und bei Mignotte Polynomen zu. Nur im Falle der sehr fein-granularen Berechnung der Zufallspolynome vom Grad 1000 führt

TABELLE 7.2.2. Vergleich der sequentiellen Implementierung der Descartes Methode mit den Systemen Maple 5.1, MuPAD 1.4.2 und MatLAB 5.3. Bis auf die Experimente mit MatLAB wurden alle Experimente auf einer UltraSPARC-IIi Workstation mit 269 MHz durchgeführt. Für die MatLAB-Experimente lag eine Windows-NT-Installation vor (Xeon Prozessor mit 550 MHz). Bei den ersten drei Systemen wurden die Nullstellen auf 8 Nachkommastellen genau bestimmt. Die MatLAB-Rechnungen wurden mit der Einstellung `double` vorgenommen. Hier sind die Zeiten eingeklammert, bei denen die Ergebnisse nicht auf 8 Nachkommastellen genau sind. Bei den Zufallspolynomen wurde hier pro Grad nur ein Testpolynom betrachtet. Da in den meisten Fällen die Laufzeit der Descartes-Methode von der anschließenden Verfeinerung der isolierenden Intervalle dominiert wird, ist sie bei dem Zufallspolynom vom Grad 400 mit 6 Nullstellen größer, als bei dem vom Grad 500 mit nur 4 Nullstellen.

Klasse/ Grad	Descartes Isolierung [s]	Descartes Gesamt [s]	Maple [s]	MuPAD [s]	MatLAB [s]
Tschebyscheff					
20	0,2	0,4	0,2	6,0	0,002
40	0,4	2,1	6,1	88,0	(0,008)
60	1,0	6,6	10,9	223,0	–
80	4,2	16,5	123,0	–	–
100	8,8	29,9	332,3	–	–
Mignotte					
20	0,1	0,1	0,8	–	0,002
40	0,6	0,7	3,4	–	0,007
60	2,8	3,0	14,7	–	0,017
80	5,3	5,6	49,1	–	0,04
100	16,8	17,2	93,4	–	0,07
Zufallspolynom					
100	0,2	0,7	15,5	–	(0,07)
200	0,6	8,0	113,7	–	(0,45)
300	1,3	16,8	390,4	–	(1,75)
400	5,5	48,9	3120,2	–	(5,8)
500	5,7	26,4	6936,8	–	(11,7)
G_n					
34	0,5	2,5	337,0	60,0	–
50	1,9	8,0	5858,0	272,0	–
70	3,5	19,9	–	831,0	–
95	15,0	55,6	–	–	–
125	29,1	119,6	–	–	–

TABELLE 7.2.3. Der Einfluss von Baum- und Knotenplanung auf die Laufzeit der parallelen Descartes Methode. Alle Experimente wurden auf auf 32 Knoten des HPCLine Systems unter Verwendung von ScaMPI durchgeführt. Die Präzision wurde optimal gewählt. Im Falle der Zufallspolynome handelt es sich um Durchschnittswerte von 50 Polynomen. Die Zeiten sind in Sekunden angegeben.

	$n = 100$		$n = 200$	
Klasse /	Knotenplanung		Knotenplanung	
Baumverteilung	aktiv	nicht aktiv	aktiv	nicht aktiv
Tschebyscheff (Grad n)				
Baumplanung	**0.16**	0.19	**1.01**	1.13
Work-Stealing	0.32	0.48	1.54	1.91
Scattering	0.24	0.29	1.60	2.00
Work-St. + Scat.	0.30	0.37	1.45	1.69
keine Vert.	0.83	1.09	4.15	5.50
Mignotte (Grad n)				
Baumplanung	**1.25**	1.54	**6.38**	7.95
Work-Stealing	2.16	2.67	9.70	12.05
Scattering	**1.24**	1.54	**6.39**	7.93
Work-St. + Scat.	2.16	2.67	9.73	12.05
keine Vert.	**1.25**	1.54	**6.38**	7.93
Zufallspolynom (Grad $10\,n$)				
Baumplanung	**0.78**	0.83	**3.22**	3.38
Work-Stealing	1.05	1.05	3.36	3.85
Scattering	0.86	0.86	**3.23**	3.52
Work-St. + Scat.	0.98	0.98	3.39	3.77
keine Vert.	0.84	0.90	**3.23**	3.62

die Baumplanung zu kürzeren Laufzeiten. Bei den Zufallspolynomen vom Grad 2000 ist die Baumplanung nur dann effizienter, wenn nicht zusätzlich die Knotenplanung benutzt wird. Falls sie benutzt wird, werden die meisten Taylor-Shifts auf höchstens 16 Prozessoren ausgeführt und sind dann so effizient, dass eine weitere Reduktion der Prozessoranzahl pro Verschiebung keinen messbaren Gewinn bringt. Die Suchbäume der Mignotte-Polynome haben die Breite 1. Dadurch ist die Abbildung der Suchknoten trivial und bei allen Verfahren gleich. Die Experimente mit den Mignotte-Polynomen zeigen jedoch, dass der Overhead der Baumplanungsmethode gering ist und dass die Benutzung von Work-Stealing in diesem Fall eine deutliche Verlangsamung mit sich bringt.

Im Falle der Tschebyscheff Polynome zeigt sich der Vorteil der Baumplanung bei breiten Suchbäumen gegenüber den anderen Verfahren. Durch die Ausnutzung der Kenntnis der Baumbreite können viele Knoten sehr effizient auf wenigen Prozessoren ausgewertet werden (vgl. Abbildung 7.2.4). Die Experimente zeigen aber auch, dass Work-Stealing und insbesondere die

Mischung von Work-Stealing und Scattering bei nicht zu feingranularen Berechnungen effizienter ist, als das senderinitiierte Scattering Verfahren. Bei den Tschebyscheff Polynomen vom Grad 100 ist das Work-Stealing Verfahren zu träge um eine schnelle Verteilung der Last zu erreichen.

Da die Knotenplanung die Anzahl der Prozessoren verringert, die denselben Taylor-Shift berechnen, wird ihr Einfluss auf die Rechenzeit von der Skalierbarkeit der Verschiebungen bestimmt. Diese wiederum hängt vom Grad des Eingabepolynoms und von der verwendeten Präzision ab. Wegen des hohen Grades der Zufallspolynome ist der Gewinn der Knotenplanung somit am kleinsten. Bei den Polynomen mit kleinerem Grad bewirkt die Knotenplanung eine größere Reduzierung der Ausführungszeiten. Eine Ausnahme bilden die Tschebyscheff-Polynome: Da die Baumplanung bereits zu einer Verringerung der jeweils einem Suchknoten zugeordneten Prozessoren führt, ist der Einfluss der Knotenplanung in diesem Fall nur gering.

Abbildung 7.2.6 zeigt die Effizienz und die Isoeffizienz des Verfahrens auf einer Cray T3E. In allen Fällen kann durch Erhöhung des Knotengrades die Effizienz der Rechnung erhöht werden. Wenn wir uns also beispielsweise auf die Klassen der Tschebyscheff oder Mignotte Polynome beschränken, scheint für die Descartes Methode sowohl die size-up als auch die speed-up Eigenschaft zu gelten. Dieses Verhalten zeigen die in Abschnitt 7.1 diskutierten Parallelisierungen nicht. Es wird ausschließlich durch die parallele Berechnung der Taylor-Shifts ermöglicht.

Wenn wir die Isoeffizienzgraphen vergleichen, wird deutlich, dass die Effizienz nicht alleine von der sequentiellen Laufzeit der Isolierung, sondern entscheidend auch von der Form des Suchbaumes abhängt. Die Effizienzfunktion der Descartes Methode erfüllt also im Allgemeinen die size-up Bedingung nicht.

Die Isoeffizienzgraphen nehmen bei den Mignotte-Polynomen die größten Werte an. Wie wir im nächsten Abschnitt sehen werden, sind die Graphen mindestens kubisch in P.

7.3. Isoeffizienz der Descartes Methode

Wir analysieren die Isoeffizienz der Descartes Methode für eine feste Präzision. Wir betrachten also nur einen Durchlauf der Methode, unabhängig davon, ob die Präzision ausreicht. Weiterhin nehmen wir an, dass das Eingabepolynom mit der gegebenen Präzision dargestellt werden kann. Da die Präzision fest ist, ist somit die maximale Koeffizientengröße des Polynoms beschränkt. Wir zeigen, dass es eine Polynomklasse gibt, bei der die Isoeffizienz kodominant mit P^2 ist. Weiterhin zeigen wir, dass die Isoeffizienz der Descartes Methode für jede Polynomklasse von $P^3 \log^2 P$ dominiert wird.

Die Analyse geht vom Taylor-Shift aus. Wie wir gesehen haben, sind die Berechnungszeiten des Taylor-Shifts diskret und die Effizienzfunktionen erfüllen die size-up und speed-up Eigenschaft. Die Isoeffizienz des Taylor-Shifts und die eines Suchknotens kann somit durch Lösen eines Gleichungssystems bestimmt werden. Hierbei setzen wir vereinfachend voraus, dass jeder Suchknoten jeweils aus drei Taylor-Shifts besteht und das grundsätzlich die effizientere der beiden Planungsmethoden benutzt wird. Schließlich wird die Isoeffizienzfunktion der Descartes Methode mit Hilfe der Isoeffizienzfunktion der Suchknoten bestimmt.

Die Variablen T, E, I beziehen sich auf die Berechnungszeit, die Effizienz und die Isoeffizienz des Taylor-Shift. Die entsprechenden Funktionen für die Suchknoten werden durch $T^{(\mathrm{v})}$, $E^{(\mathrm{v})}$, $I^{(\mathrm{v})}$ bezeichnet, und für die vollständige Descartes Methode benutzen wir $T^{(\mathcal{D})}, E^{(\mathcal{D})}, I^{(\mathcal{D})}$. Wie in Kapitel 2 werden durch Über- bzw. Unterstreichungen obere und untere Schranken kenntlich gemacht.

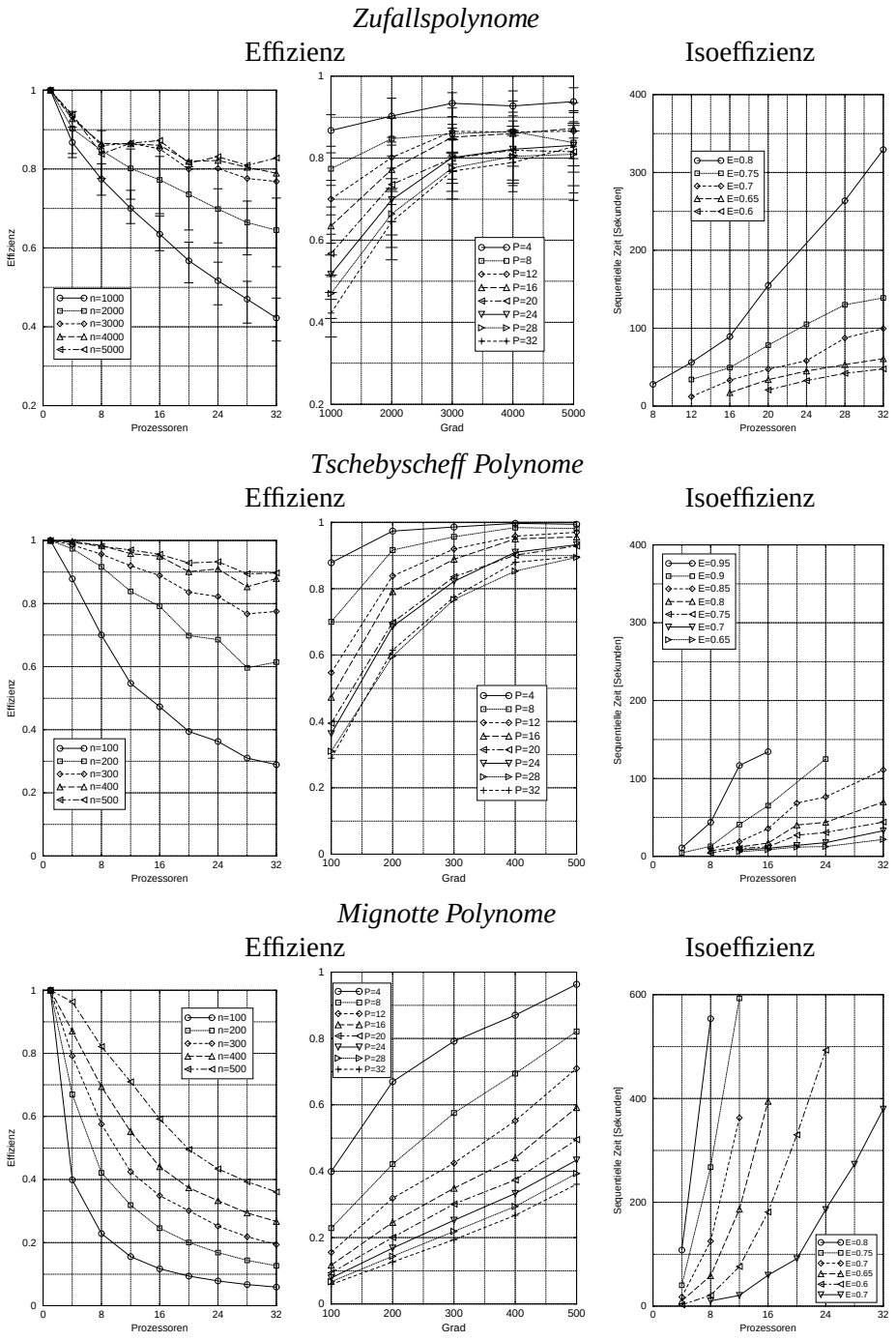


ABBILDUNG 7.2.6. Effizienz und Isoeffizienz der Descartes Methode im Experiment. Im Falle der Zufallspolynome wurden jeweils 50 Experimente durchgeführt. Die vertikalen Markierungen zeigen hier die Schwankungsbereiche der Messungen an. Die Experimente wurden auf der Cray T3E durchgeführt.

7.3.1. Isoeffizienz des Taylor-Shifts. Wir zeigen, dass die Isoeffizienz der Taylor-Shift Berechnung kodominant mit P^2 ist. Es wurde bereits in Abschnitt 3.2 gezeigt, dass die Laufzeit von Pyramidenberechnungen mit konstanten Knotenbearbeitungszeiten P^2 dominiert. Im Folgenden zeigen wir, dass die Berechnung des Taylor-Shifts in Bezug auf die Isoeffizienz optimal ist.

THEOREM 7.3.1. *Es sei* $\overline{e}(s,\,P) := \frac{s^2 P}{s^2 P + P - 1}$. *Dann sind die Werte* $I_e(P)$ *und* $\overline{I}_e(P)$ *für jedes* $e \in (0,\,\overline{e}\,(s,\,P))$ *definiert. Weiterhin sei* $\overline{e}(s) := \frac{s^2}{s^2+1}$. *Dann sind die Werte* $I_e(P)$ *und* $\overline{I}_e(P)$ *für alle* $e \in (0,\,\overline{e}(s))$ *und* $P \in \mathbb{N}$ *definiert und kodominant mit* P^2.

BEWEIS. Die erste Aussage gilt, da $\overline{E}$ und $\underline{E}$ monoton in n steigen und

$$\lim_{n\to\infty} E(n,P) = \lim_{n\to\infty} \underline{E}(n,\,P) = \lim_{n\to\infty} \overline{E}(n,\,P) = \overline{e}\,(s,\,P) \quad .$$

Die zweite Aussage gilt wegen $\lim_{P\to\infty} \overline{e}(s,P) = \overline{e}(s)$.

Da die Berechnung des Taylor-Shifts eine Pyramidenberechnung mit konstanter Aufgabenbearbeitungszeit ist, gilt wegen Theorem 3.2.5, $I_e(P) \succeq P^2$. Es bleibt also noch $\overline{I}_e(P) \preceq P^2$ zu zeigen. Hierfür sei $e \in (0,\,\overline{e}\,(s))$. Die Ungleichung $\underline{E}(n,\,P) \geq e$ ist äquivalent zu

$$0 \geq \underbrace{\left(s^2P + P - 1 - \frac{s^2P}{e}\right)\frac{t}{2}}_{=:\,A}\, n^2 + \underbrace{\left(\frac{e-1}{e}\frac{s^2Pt}{2} + (sP-1)sLP\right)}_{=:\,B}\, n + \underbrace{(2sP+P-4)\,o\,s^2P^2}_{=:\,C} \quad .$$

Wegen $e < \overline{e}(s)$ ist A negativ. Folglich gilt

$$\underline{E}(n,\,P) \geq e \;\text{ genau dann wenn }\; n \geq \left\lceil -\frac{B}{2A} + \sqrt{\frac{B^2}{4A^2} - \frac{C}{A}}\;\right\rceil =: n_0 \quad .$$

Wir erhalten also $\overline{I}_e(P) = T(n_0,\,1)$. Da $\frac{B}{A} \preceq P$ und $\frac{C}{A} \preceq P^2$ gilt $n_0 \preceq P$. Nun impliziert $T(n,1) \sim n^2$ die Behauptung $\overline{I}_e(P) \preceq P^2$. □

7.3.2. Isoeffizienz der Knotenberechnungen. Für die Analyse gehen wir bei der Knotenberechnung davon aus, dass stets die effizientere der beiden Berechnungsmethoden A und B benutzt wird. Die Knotenberechnung ist somit ein Polyalgorithmus im Sinne von Theorem 2.2.18. Um dieses Theorem anwenden zu können, benötigen wir zunächst Aussagen über die Isoeffizienz der beiden Methoden. Da die Beweise hierfür analog zum Beweis von Theorem 7.3.1 verlaufen, werden sie nicht im einzelnen ausformuliert.

Wir bezeichnen mit $T^{(\mathrm{A})}$, $E^{(\mathrm{A})}$, $I^{(\mathrm{A})}$ die Laufzeit, Effizienz und Isoeffizienz für Methode A und mit $T^{(\mathrm{B})}$, $E^{(\mathrm{B})}$, $I^{(\mathrm{B})}$ die entsprechenden Werte für Methode B. Für die Laufzeiten erhalten wir (am Beispiel der unteren Schranken):

$$\begin{aligned}
\underline{T}^{(\mathrm{A})}(n,\,P) &= \underline{T}\left(n,\,\left\lfloor\frac{P}{2}\right\rfloor\right) + \underline{T}(n,\,P) \\
\underline{T}^{(\mathrm{B})}(n,\,P) &= \max\left(\underline{T}(n,\,\left\lfloor\frac{P}{3}+\frac{1}{2}\right\rfloor,\,2\,\underline{T}(n,\,P-\left\lfloor\frac{P}{3}+\frac{1}{2}\right\rfloor\right) \quad .
\end{aligned}$$

Hiermit erhalten wir die oberen Schranken für die Effizienzen der Methoden

$$\overline{E}^{(\mathrm{A})}\,(n,\,P) := \frac{3\,T(n,\,1)}{\underline{T}^{(A)}\,(n,\,P)} \;\text{ und }\; \overline{E}^{(\mathrm{B})}\,(n,\,P) := \frac{3\,T(n,\,1)}{\underline{T}^{(\mathrm{B})}\,(n,\,P)} \quad .$$

und schließlich gemäß Theorem 2.2.3 untere Schranken für die Isoeffizienzfunktionen $I_e^{(\mathrm{A})}(P)$ und $I_e^{(\mathrm{B})}(P)$. Analoge Beziehungen gelten für die oberen Schranken der Laufzeiten und Isoeffizienzen sowie für die unteren Schranken der Effizienzen. Damit sind die Effizienzschranken des Suchknotens gegeben durch

$$\begin{aligned} \underline{E}^{(\mathrm{v})}(n,\, P) &= \min\left(\underline{E}^{(\mathrm{A})}(n,\, P),\, \underline{E}^{(\mathrm{B})}(n,\, P)\right) \text{ und} \\ \overline{E}^{(\mathrm{v})}(n,\, P) &= \min\left(\overline{E}^{(\mathrm{A})}(n,\, P),\, \overline{E}^{(\mathrm{B})}(n,\, P)\right) \ . \end{aligned}$$

THEOREM 7.3.2. *Es sei $\alpha \in \{A,\, B\}$ eine Planungsmethode für Suchknoten. Dann haben die Funktionen $\underline{E}^{(\alpha)}$ und $\overline{E}^{(\alpha)}$ die speed-up und die size-up Eigenschaft. Ferner sind für $e \in (0,\, \overline{e}(s))$ die Funktionen $I_e^{(\alpha)}(P)$, $\overline{I}_e^{(\alpha)}(P)$ und $\underline{I}_e^{(\alpha)}(P)$ für alle $P \in \mathbb{N}$ definiert und kodominant mit P^2.*

BEWEIS. Da die Effizienzfunktion des Taylor-Shifts die size-up und speed-up Bedingungen erfüllen, gelten diese Bedingungen auch für die Effizienzfunktionen der beiden Planungsmethoden. Die Isoeffizienzfunktionen sind definiert, da für $e < \overline{e}(s)$ die Funktion $I_e(P)$ definiert ist und ein Suchknoten stets mit höherer Effizienz ausgeführt wird, als ein einzelner Taylor-Shift. Für die Herleitung der Kodominanz der Funktionen mit P^2 betrachtet man analog zum Beweis von Theorem 7.3.1 für $\alpha \in \{\mathrm{A},\, \mathrm{B}\}$ die Ungleichung $\underline{E}^{(\alpha)}(n,\, P) \geq e$ und leitet auf diese Weise die Beziehung $\overline{I}_e^{(\alpha)}(P) \preceq P^2$ her. Ebenso erhält man aus $\overline{E}^{(\alpha)}(n,\, P) \geq e$ die Aussage $\underline{I}_e^{(\alpha)}(P) \succeq P^2$. □

THEOREM 7.3.3. *Die Effizienzschranken der Berechnung eines Suchknotens, $\underline{E}^{(\mathrm{v})}(n,\, P)$ und $\overline{E}^{(\mathrm{v})}(n,\, P)$, haben die speed-up und die size-up Eigenschaft. Die Isoeffizienzschranken $\underline{I}_e^{(\mathrm{v})}(P)$ und $\overline{I}_e^{(\mathrm{v})}(P)$ sind kodominant mit P^2.*

BEWEIS. Die Behauptung folgt aus den Theoremen 2.2.18 und 7.3.2. □

7.3.3. Isoeffizienz der Descartes Methode. Wir bezeichnen mit $\mathcal{D}(A,\, P)$ den phasenparallelen Plan, der bei der Nullstellenisolierung von Polynom A auf P Prozessoren generiert wird. Die Schranken für die Knotenberechnungszeiten liefern uns Schranken für die gesamte Isolierungszeit $T^{(\mathcal{D})}(A,\, P)$, für die Effizienz $E^{(\mathcal{D})}(A,\, P)$ der Isolierung und für ihre Isoeffizienz $I_e^{(\mathcal{D})}(P)$. Zunächst untersuchen wir die Isoeffizienz von Isolierungen mit beschränkter Suchbaumbreite und -größe.

THEOREM 7.3.4. *Es sei $I_e^{(\mathcal{D})}$ die Isoeffizienzfunktion der Descartes Methode für die Menge der Polynome, deren Suchbaumbreite höchstens N ist. Dann gilt $P^2 \preceq I_e^{(\mathcal{D})}(P)$.*

BEWEIS. Die Breite (Definition 4.4.1) von $\mathcal{D}(A,\, P)$ ist durch die Suchbaumbreite von A beschränkt. Für die Analyse des asymptotischen Verhaltens von $I_e^{(\mathcal{D})}(P)$ betrachten wir den Fall $P > 2N$. Da $\overline{E}^{(\mathrm{v})}$ die size-up und die speed-up Eigenschaften erfüllt, gilt $\underline{I}_e^{(\mathrm{v})}(\lfloor P/N \rfloor) \leq \underline{I}_e^{(\mathcal{D})}(P)$ (Theorem 4.4.4). Damit folgt

$$P^2 \sim (P/N)^2 \sim (\lfloor P/N \rfloor)^2 \sim \underline{I}_e^{(\mathrm{v})}(\lfloor P/N \rfloor) \leq \underline{I}_e^{(\mathcal{D})}(P) \leq I_e^{(\mathcal{D})}(P) \ .$$

□

THEOREM 7.3.5. *Es sei $I_e^{(\mathcal{D})}$ die Isoeffizienzfunktion für Polynome, die Suchbäume mit höchstens N Knoten generieren. Dann gilt $I_e^{(\mathcal{D})}(P) \sim P^2$.*

BEWEIS. Wegen der size-up und speed-up Eigenschaft von $\underline{E}^{(\mathrm{v})}$ gilt $\overline{I}_e^{(\mathcal{D})}(P) \leq N\,\overline{I}_e^{(\mathrm{v})}(P)$ (Theorem 4.4.5). Folglich gilt

$$I_e^{(\mathcal{D})}(P) \leq \overline{I}_e^{(\mathcal{D})}(P) \leq N\,\overline{I}_e^{(\mathrm{v})}(P) \sim N\,P^2 \sim P^2 \ .$$

Da die Breite eines Suchbaumes mit höchstens N Knoten ebenfalls beschränkt ist, erhalten wir mit Theorem 7.3.4 die Behauptung $I_e^{(\mathcal{D})}(P) \succeq P^2$. □

Die Polynome $x^n - 2$, $n \geq 2$, haben beschränkte Koeffizienten und Suchbäume mit höchstens 3 Knoten; sie bilden also eine Klasse, auf der die Descartes Methode in der Tat eine quadratische Isoeffizienz hat.

THEOREM 7.3.6. *Es sei $I_e^{(\mathcal{D})}(P)$ die Isoeffizienzfunktion von Polynomen, deren Suchbäume nur einen Suchknoten pro Baumebene haben und deren Suchbaumhöhe den Grad dominiert. Dann gilt $P^3 \preceq I_e^{(\mathcal{D})}(P)$.*

BEWEIS. Es sei X eine Klasse solcher Polynome. Dann gibt es ein $a > 0$, so dass für alle $A \in X$, $a \deg A \leq |\mathcal{D}(A,\,1)|_1$. Somit gilt $T^{(\mathcal{D})}(A,\,1) = |\mathcal{D}(A,\,1)|_1\,T^{(\mathrm{v})}(\deg A,\,1) \geq a \deg A\,T^{(\mathrm{v})}(\deg A,\,1)$ für alle $A \in X$. Es sei $A \in X$ so dass $\underline{I}_e^{(\mathcal{D})}(P) = T^{(\mathcal{D})}(A,\,1)$ und es sei $n = \deg A$. Da jeder Suchknoten dieselbe Effizienz hat, gilt

$$\overline{E}^{(\mathrm{v})}(n,\,P) = \overline{E}^{(\mathcal{D})}(A,\,P) \geq e \ .$$

Wegen der size-up Eigenschaft von $\overline{E}^{(\mathrm{v})}$ erhalten wir mit Theorem 2.2.9 die Aussage $T^{(\mathrm{v})}(n,\,1) \geq \underline{I}_e^{(\mathrm{v})}(P)$. Da jeder Knoten aus drei Taylor-Shifts besteht, impliziert Theorem 7.2.1, dass $n \geq \frac{1}{2} + \sqrt{\frac{1}{4} + 2\frac{\underline{I}_e^{(\mathrm{v})}(P)}{3\,t}}$. Da $\underline{I}_e^{(\mathrm{v})}(P) \succeq P^2$, gibt es $b,\,c > 0$ mit $n \geq b\,P$ und $T^{(\mathrm{v})}(n,\,1) \geq \underline{I}_e^{(\mathrm{v})}(P) \geq c\,P^2$. Insgesamt erhalten wir

$$I_e^{(\mathcal{D})}(P) \geq \underline{I}_e^{(\mathcal{D})}(P) = T^{(\mathcal{D})}(A,\,1) \geq a\,n\,T^{(\mathrm{v})}(n,\,1) \geq a \cdot b\,P \cdot c\,P^2 \succeq P^3 \ .$$

□

Die Koeffizienten von Mignotte Polynomen sind beschränkt und die Höhe der Suchbäume dominiert ihren Grad. Experimente zeigen, dass die Suchbaumhöhe etwa das 1,2-fache des Grades ist und dass die Suchbäume Pfade sind. Durch die Auswertung der Isoeffizienzfunktion mit den für die Cray T3E relevanten LogP-Parametern ($L = 20\,\mu$sec., $o = 0.1$ msec, $t = 25\,\mu$sec , und $s = 3$) können wir die gemessenen Isoeffizienzen in Abbildung 7.2.2 voraussagen. Die Abbildung 7.3.1 zeigt die gute Übereinstimmung der theoretischen und der gemessenen Werte.

Um die Isoeffizienz der Descartes Methode für beliebige Klassen bestimmen zu können, benutzen wir einen allgemeinen Zusammenhang zwischen Polynomgrad und der Anzahl der Suchknoten, der bei beschränkter Koeffizientengröße besteht.

LEMMA 7.3.7. *Wenn die reellen Nullstellen eines quadratfreien Polynoms mit der Descartes Methode isoliert werden (Fließkomma-Intervallarithmetik mit fester Mantissen- und Exponentenlänge) und das Polynom den Grad n hat, dann wird die Anzahl der Suchknoten von $n\,(\log n)^2$ dominiert.*

BEWEIS. Wegen der festen Mantissen- und Exponentenlänge sind die Koeffizienten des Eingabepolynoms durch eine Konstante D beschränkt. Es sei A ein Polynom und es sei Υ der generierte Suchbaum. Der Suchbaum habe die Tiefe K. Für $1 \leq k \leq K$ sei $h(k)$ die Anzahl der Knoten auf Suchbaumebene k. Da nach [**85**, Theorem 48] $k\,h(k) \preceq n \log(n^2 D) \preceq n \log n$ und

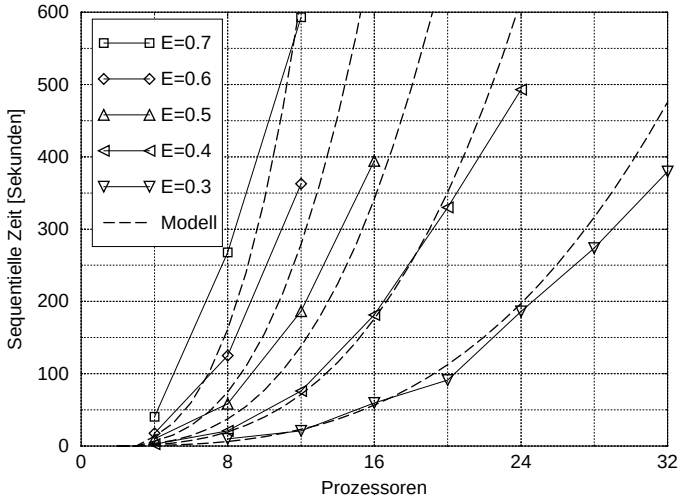


ABBILDUNG 7.3.1. Der Vergleich der Isoeffizienz von Mignotte Polynomen im Modell mit gemessenen Isoeffizienzgraphen zeigt eine gute Übereinstimmung zwischen Modell und Realität.

$K \preceq n\log(n^2 D) \preceq n\log n$ gilt, können wir die Anzahl der Knoten von Υ beschränken. Es sei H_K die K-te harmonische Zahl. Dann gibt es ein $C > 0$ mit

$$\begin{aligned}\sum_{k=1}^{K} h(k) &\leq \sum_{k=1}^{K} \frac{C\, n\log(n)}{k} \\ &\leq C\, n\, \log(n)\, H_K \\ &\preceq n\log(n)\, \log\left(n\, \log(n)\right) \\ &\preceq n\,(\log n)^2 \quad .\end{aligned}$$

□

THEOREM 7.3.8. *Die Isoeffizenzfunktion $I_e^{(\mathcal{D})}(P)$ der Descartes Methode wird von $P^3\,(\log P)^2$ dominiert.*

BEWEIS. Es sei P eine Prozessoranzahl und es sei A ein Polynom mit Grad n so dass $\overline{I}_e^{(\mathcal{D})}(P) = T^{(\mathcal{D})}(A,\,1)$. Ein solches Polynom existiert, da die sequentiellen Rechenzeiten der Descartes Methode diskret sind. Da die Wurzel des Suchbaumes von A von P Prozessoren berechnet wird, ist ihre Effizienz höchstens e, also $\underline{E}^{(\mathrm{v})}(n,\,P) \leq e$. Wegen der size-up Eigenschaft von $\underline{E}^{(\mathrm{v})}$ erhalten wir mit Theorem 2.2.1, dass $T^{(\mathrm{v})}(n,\,1) \leq \overline{I}_e^{(\mathrm{v})}(P)$. Folglich gilt $n \leq \frac{1}{2} + \sqrt{\frac{1}{4} + 2\frac{\overline{I}_e^{(\mathrm{v})}(P)}{3\,t}}$. Mit Theorem 7.3.3 erhalten wir $\overline{I}_e^{(\mathrm{v})} \preceq P^2$, so dass es positive Zahlen a und b gibt, mit $n \leq aP$ und $T^{(\mathrm{v})}(n,\,1) \leq bP^2$. Wegen Lemma 7.3.7 gibt es positive Zahlen $c,\,c'$ so dass $|\mathcal{D}(A,\,1)|_1 \leq cn\,(\log(n))^2 \leq caP\,(\log(aP))^2 \leq c'\,P\,(\log P)^2$. Insgesamt erhalten wir

$$I_e^{(\mathcal{D})}(P) \leq \overline{I}_e^{(\mathcal{D})}(P) = T^{(\mathcal{D})}(A,\,1) = |\mathcal{D}(A,\,1)|_1\, T^{(\mathrm{v})}(n,\,1) \leq c'P\,(\log P)^2 \cdot bP^2 \preceq P^3\,(\log P)^2 \quad .$$

□

Literaturverzeichnis

[1] A.K. Amoura, E. Bampis, C. Kenyon, and Y. Manoussakis. Scheduling independent multiprocessor tasks. In R. Burkhard and G. Woeginger, editors, *Proc. of the 5thEuropean Symposium on Algorithms (ESA)*, volume 1284 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1997.

[2] J. N. C. Árabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome: Parallel programming in a heterogeneous multi-user environment. In *Proc. of the 10th International Parallel Processing Symposium (IPPS)*, 1996.

[3] B. Baker, E. Coffman, and R. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 9(4):846–855, Nov. 1980.

[4] M. Barnett, T. Decker, and W. Krandick. Power series expansion of the roots of a secular equation containing symbolic elements: Computer algebra and Moseley's law. *Journal of Chemical Physics, to appear*, 2001.

[5] K.P. Belkhale and P. Banerjee. Partitionable independent task scheduling problem. In *Proc. of the 1990 International Conference on Parallel Processing*, volume 1, pages 72–75, 1990.

[6] N. Biggs. *Algebraic Graph Theory, Second Edition.* Cambridge University Press, 1974/1993.

[7] W. Blochinger, W. Küchlin, and A. Weber. The distributed object-orientated threads system DOTS. In *Proc. of the 5th International Symp. on Parallel Algorithms for Irregularly Structured Problems*, pages 206–217, 1998.

[8] R D. Blumhofe, C F Joerg, B. C Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proc. of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (PPOPP '95)*, pages 207–216, 1995.

[9] R. D. Blumhofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proc. of the 25th IEEE Symp. on Foundations of Computer Science (FOCS)*, page 356-368. IEEE, 1994.

[10] S. Bock and O. Rosenberg. Dynamic load balancing strategies for planning production processes in heterogeneous networks. In *Proceedings of the Conference of the Institute of Operations Research and Management Sciences, INFORMS 2000*, pages 951–955, 2000.

[11] C. Boeres, G. Chochia, and P. Thanisch. On the scope of applicability of the ETF algorithm. In A. Ferreira and J. Rolim, editors, *Proc. of the 2nd International Symp. on Parallel Algorithms for Irregularly Structured Problems*, number 980 in Lecture Notes in Computer Science, pages 159–164. Springer, 1995.

[12] P. G. Bradford. Efficient parallel dynamic programming. In *Proc. of the 30th Allerton Conference on Communication, Control and Computation*, pages 185–194, 1992.

[13] K. Brockmann and T. Decker. A parallel tabu search algorithm for short term lot sizing and scheduling in flexible flow line environments. In *Proc. of the 16th International Conference on CAD/CAM, Robotics and Factories of the Future*, pages 843–850, 2000.

[14] T. Bubeck, M. Hiller, W. Küchlin, and W. Rosenstiel. Distributed symbolic computation with DTS. In *Proc. of the 2nd International Symp. on Parallel Algorithms for Irregularly Structured Problems*, pages 231–248, 1995.

[15] N. Carriero, D. Gelernter, T. G. Mattson, and A. H. Sherman. The Linda alternative to message-passing systems. *Parallel Computing*, 20(4):633–655, 1994.

[16] B.W. Char, K. O. Geddes, G.H. Gonnet, B.L. Leong, M.B. Monagan, and S.M. Watt. *Maple V Language Reference Manual.* Springer-Verlag, New York, 1991.

[17] J. Chen and A. Miranda. A polynomial time approximation scheme for general multiprocessor job scheduling. In *Proc. of the 31st ACM Symp. on Theory of Computing (STOC)*, pages 418–427. ACM, 1999.

[18] A.A. Chien, J. Dolby, B. Gangul, V. Karamcheti, and X. Zhang. Evaluating high level parallel programming support for irregular applications in ICC++. *Software–Practice and Experience*, 28(11):1213–1243, Sept. 1998.

[19] A.A. Chien and J.T. Dolby. *Parallel programming using C++*, chapter ICC++, pages 343–382. MIT Press, Massachusetts, 1996.

[20] R. S. Chin and S. T. Chanson. Distributed object-based programming systems. *ACM Comput. Surv.*, 23(1):91–124, March 1991.
[21] P. Chrétienne. A polynomial algorithm to optimally schedule tasks over a virtual distributed system under tree-like precedence constraints. *European Journal on Operations Research*, 43:225–230, 1989.
[22] P. Chrétienne. Task scheduling with interprocessor communication delays. *European Journal on Operations Research*, 57:348–354, 1992.
[23] P. Chrétienne. Complexity of tree-scheduling with interprocessor communication delays. *Discrete Applied Mathematics*, 1:1–10, 1994.
[24] P. Chrétienne and C. Picouleau. *Scheduling with communication delays: A survey*, chapter 4, pages 65–90. Wiley & Sons, 1995.
[25] J.Y. Colin and P. Chrétienne. C.P.M. Scheduling with small communication delays and task duplication. *Operations Research*, 39:680–684, 1991.
[26] G. E. Collins. The computing time of the Euclidean algorithm. *SIAM Journal on Computing*, 3(1):1–10, 1974.
[27] G. E. Collins and A. G. Akritas. Polynomial real root isolation using Descartes' rule of signs. In R. D. Jenks, editor, *Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, pages 272–275. ACM, 1976.
[28] G. E. Collins, J. R. Johnson, and W. Küchlin. Parallel real root isolation using the coefficient sign variation method. In R. E. Zippel, editor, *Proc. of Second International Workshop of Computer Algebra and Parallelism*, pages 71–87. Springer, 1992.
[29] G.E. Collins et al. `SACLIB` User's Guide. Technical Report 93-19, Research Institute for Symbolic Computation, RISC-Linz, Johannes Kepler University, A-4040 Linz, Austria, 1993.
[30] G.E. Collins and W. Krandick. An efficient algorithm for infallible complex root isolation. In P.S. Wang, editor, *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 189–194. ACM, 1992.
[31] F. Comellas. (degree,diameter)-graphs. http://www-mat.upc.es/grup_de_grafs/table_g.html.
[32] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. ISBN 0-262-03141-8. MIT Press, 1989.
[33] A. Corradi, L. Leonardi, and F. Zambonelli. Diffusive load-balancing policies for dynamic applications. *IEEE Concurrency*, 7(1):22–31, 1999.
[34] A. Corradi, L. Leonardi, and F. Zambonelli. Parallel objects migration: A fine grained approach to load distribution. *Journal of Parallel and Distributed Computing*, 60:48–71, 2000.
[35] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12. ACM, May 1993.
[36] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7:279–301, 1989.
[37] T. Decker and R. Diekmann. Mapping of coarse-grained applications onto workstation-clusters. In *Proc. of the 5th EUROMICRO Workshop on Parallel and Distributed Processing (PDP'97)*, pages 5–12, 1997.
[38] T. Decker and W. Krandick. Parallel real root isolation using the Descartes method. In P. Banerjee, V.K. Prasanna, and B.P. Sinha, editors, *Proc. of the 6th International Conference on High Performance Computing*, volume 1745 of *Lecture Notes in Computer Science*, pages 261–268. Springer, 1999.
[39] T. Decker and W. Krandick. Isoefficiency and the parallel Descartes method. In *Proceedings of the 7th Rhine Workshop on Computer Algebra (RWCA'00)*, 2000.
[40] T. Decker, R. Lüling, and S. Tschöke. A distributed load balancing algorithm for heterogeneous parallel computing systems. In H. R. Arabnia, editor, *Proc. of the 1998 Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, volume 2, pages 933–940. CSREA Press, July 1998.
[41] Y. Denneulin, J. M. Geib, and J. F. Mehaut. A multithreaded-based methodology to solve irregular problems. In *Proc. of the POC'96*, 1996. http://www.lifl.fr/ mehaut/publis/POC96.ps.gz.
[42] R. Diekmann. *Load balancing strategies for data parallel applications*. PhD thesis, Universität Paderborn, 1998.
[43] R. Diekmann, A. Frommer, and B. Monien. Nearest neighbor load balancing on graphs. In G. Bilardi, G. Italiano, A. Piertracaprina, and G. Pucci, editors, *Proc. of the European Symposium on Algorithms (ESA'98)*, volume 1461 of *Lecture Notes in Computer Science*, pages 429–440. Springer Verlag, 1998.
[44] R. Diekmann, A. Frommer, and B. Monien. Efficient schemes for nearest neighbor load balancing. *Parallel Computing*, 25(7):789–812, 1999.

[45] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw. Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM. *Parallel Computing*, 26(12):1555–1581, 2000.
[46] J. Dongarra, H. Meuer, and E. Strohmaier. Top 500 Supercomputer Sites. http://www.top500.org/, 2000.
[47] M. Drozdowski. Scheduling multiprocessor tasks – An overview. *European Journal of Operational Research*, 94:215–230, 1996.
[48] J. Du and Y-T. Leung. Complexity of scheduling parallel tasks systems. *SIAM Journal of Discrete Mathematics*, 2(4):473–487, 1989.
[49] A. Edelman and E. Kostlan. How many zeros of a random polynomial are real? *Bulletin (New Series) of the American Mathematical Society*, 32(1):1–37, January 1995.
[50] R. Elsässer, A. Frommer, B. Monien, and R. Preis. Optimal and alternating-direction loadbalancing schemes. In *Proc. of the 5th EuroPar Conference*, volume 1685 of *Lecture Notes in Computer Science*, pages 280–290. Springer, 1999.
[51] R. Elsässer, R. Královič, and B. Monien. Scalable sparse topologies with small spectrum. In *Proc. of the 18th Symp. on Theoretical Aspects of Computer Science (STACS)*, 2001. to appear.
[52] G. Fertin, A. Raspaud, H. Schröder, O. Sýkora, and I. Vřto. Diameter of the Knödel graph. In U. Brandes and D. Wagner, editors, *Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, volume 1253 of *Lecture Notes in Computer Science*, pages 149–160. Springer, 2000.
[53] F. Feschet, S. Miguet, and L. Perroton. Parlist: A parallel data structure for dynamic load balancing. *Journal of Parallel and Distributed Computing*, 51(2):114–135, 1998.
[54] R. Friedman, M. Goldin, A. Itzkovitz, and A. Schuster. Millipede:Easy parallel programming in available distributed environments. *Software–Practice and Experience*, 27(8):929–965, Aug. 1997.
[55] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, volume 33 of *SIGPLAN Notices*, pages 212–223, 1998.
[56] C. Fu and T. Yang. Run-time compilation for sparse matrix computations. In *Proceedings of the ACM International Conference on Supercomputing*, pages 237–244. ACM, 1996.
[57] G-L.Park, B. Shirazi, and J. Marquis. DFRN: A new approach on duplication based scheduling for distributed memory multiprocessor systems. In *Proc. of the 8th International Parallel Processing Symposium (IPPS)*, pages 157–166, 1994.
[58] G-L.Park, B. Shirazi, and J. Marquis. Comparative study of static scheduling with task duplication for distributed systems. In G. Bilardi, A. Ferreira, R. Lüling, and J. Rolim, editors, *Proc. of the 4th International Symp. on Parallel Algorithms for Irregularly Structured Problems*, volume 1253 of *Lecture Notes in Computer Science*, pages 123–134. Springer, 1997.
[59] F. Galilée, J.-L. Roch, G. Cavalheiro, and M. Doreille. Athapascan-1: On-line building data flow graph in a parallel language. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 88–95. IEEE, 1998.
[60] M. Garey and R. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM Journal on Computing*, 4(2):187–200, 1975.
[61] M.R. Garey and D.S. Johnson. *Computers and Intractability: A guide to the theory of NP-completeness*. Freeman, San Francisco, 1979.
[62] M. Gengler. An introduction to parallel and dynamic programming. In A. Ferreira and P. Paradalos, editors, *Solving Combinatorial Optimization Problems in Parallel*, volume 1054 of *Lecture Notes in Computer Science*, pages 87–114, 1996.
[63] A. Gerasoulis and T. Yang. Static scheduling of parallel programs for message passing architectures. In *Proc. of the CONPAR/VAPP'92*, pages 601–611, 1992.
[64] B. Ghosh, S. Muthukrishnan, and M.H. Schultz. First and second order diffusive methods for rapid, coarse, distributed load balancing. In *ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 72–81, 1996.
[65] A. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel and Distributed Technology, Special Issue on Parallel and Distributed Systems: From Theory to Practice*, 1(3):12–21, Aug. 1993.
[66] A. S. Grimshaw. Easy to use object-oriented parallel programming with Mentat. *IEEE Computer*, pages 39–51, May 1993.
[67] A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):502–520, 1997.

[68] A. Gupta and V. Kumar. The scalability of FFT on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):922–932, 1993.
[69] M. C. Heydemann, N. Marlin, and S. Perennes. Cayley graphs with complete rotations. Technical Report 1155, L.R.I. Orsay, 1997.
[70] Y.F. Hu and R.J. Blake. An improved diffusion algorithm for dynamic load balancing. *Parallel Computing*, 25(4):417–444, 1999.
[71] Y.F. Hu, R.J. Blake, and D.R. Emerson. An optimal migration algorithm for dynamic load balancing. *Concurrency – Practice & Experience*, 10(6):467–483, 1998.
[72] S.-H. S. Huang, H. Liu, and V. Viswanathan. A sublinear parallel algorithm for some dynamic programming problems. *Theoretical Computer Science*, 106:361–371, 1992.
[73] C.-C. Hui and S. T. Chanson. Flexible and extensible load balancing. *Software–Practice and Experience*, 27(11):1283–1306, Nov. 1997.
[74] J-J. Hwang, Y-C. Chow, F.D. Anger, and C-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal of Computing*, 18(2):244–257, 1989.
[75] A. Jakoby and R. Reischuk. The complexity of scheduling problems with communication delays on trees. In *Proc. of the 3rd Scandinavian Workshop on Algorithm Theory*, pages 165–177, 1992.
[76] A. Jakoby and R. Reischuk. Scheduling trees with communication delays. Technical report, Med. Universität zu Lübeck, 1997.
[77] K. Jansen and L. Porkolab. Linear time approximation schemes for scheduling malleable parallel tasks. In *SIAM Symp. on Discrete Algorithms (SODA)*, pages 490–498, 1999.
[78] J.R. Johnson. *Algorithms for Polynomial Real Root Isolation.* PhD thesis, The Ohio State University, 1991.
[79] J.R. Johnson. Algorithms for polynomial real root isolation. In B.F. Caviness and J.R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, Texts and Monographs in Symbolic Computation, pages 269–299. Springer, 1998.
[80] H. Jung, L.M. Kirousis, and P. Spirakis. Lower bounds and efficient algorithms for multiprocessor scheduling of directed acyclic graphs with communication delays. *Information and Computation*, 105:94–104, 1993.
[81] L.V. Kalé and S. Krishnan. *Parallel programming using C++*, chapter CHARM++, pages 175–214. MIT Press, Massachusetts, 1996.
[82] G. Karypis and V. Kumar. Efficient parallel mappings of a dynamic programming algorithm: A summary of results. In *Proc. of the 7th International Parallel Processing Symposium (IPPS)*, pages 563–568. IEEE Computer Society Press, 1993.
[83] W. Knödel. New gossips and telephones. *Discrete Mathematics*, 13:95, 1975.
[84] D.E. Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison-Wesley, 1st edition, 1969.
[85] W. Krandick. Isolierung reeller Nullstellen von Polynomen. In J. Herzberger, editor, *Wissenschaftliches Rechnen*, pages 105–154. Akademie Verlag, Berlin, 1995.
[86] R. Krishnamurti and E. Ma. The processor partitioning problem in special-purpose partitionable systems. In *Proc. of the 1988 International Conference on Parallel Processing*, volume 1, pages 434–443, 1988.
[87] R. Krishnamurti and B. Narahari. An approximation algorithm for preemptive scheduling on parallel-task systems. *SIAM Journal on Discrete Mathematics*, 8(4):661–669, Nov. 1995.
[88] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71(1):95–132, 1990.
[89] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin/Cummings, Redwood City, CA, USA, 1994.
[90] V. Kumar, V.Nageshwara Rao, and K. Ramesh. Parallel depth first search on the ring architecture. In *Proc. of the International Conference on Parallel Processing*, pages 128–132, 1988.
[91] V. Kumar and V. Singh. Scalability of parallel algorithms for the all-pairs shortest-path problem. *Journal of Parallel and Distributed Computing*, 13:124–138, 1991.
[92] E. Lander, J. P. Mesirov, and W. Taylor. Study of protein sequence comparison metrics on the connection machine cm-2. *Journal of Supercomputing*, 3:255–269, 1989.
[93] M. L.Dowling. A fast parallel Horner algorithm. *SIAM Journal on Computing*, 19(1):133–142, 1990.
[94] G. Lewandowski, A. Condon, and E. Bach. Asynchronous analysis of parallel dynamic programming algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 7(4):425–437, 1996.

[95] W. Ludwig and P. Tiwari. Scheduling malleable and nonmalleable parallel tasks. In *SIAM Symp. on Discrete Algorithms (SODA)*, pages 167–175, 1994.
[96] R. Lüling. *Lastverteilungsverfahren zur effizienten Nutzung paralleler Systeme.* PhD thesis, Universität Paderborn, 1997.
[97] R. Lüling and B. Monien. A dynamic distributed load balancing algorithm with provable good performance. In *Proc. of the 5th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 164–173, 1993.
[98] B. M. Maggs, L. R. Matheson, and R. E. Tarjan. Models of parallel computation: A survey and synthesis. In *Proc. of the 28th Annual Hawaii International Conference on System Sciences*, pages 61–70, 1995.
[99] N. R. Mahapatra and S. Dutt. Scalable global and local hashing strategies for duplicate pruning in parallel A^* graph search. *IEEE Transactions on Parallel and Distributed Systems*, 8(7):738–756, 1997.
[100] C. Mittermaier. Parallel algorithms in constructive algebraic geometry. Master's thesis, Johannes Keppler University, Linz, Austria, 2000.
[101] G. Mounie, C. Rapine, and D. Trystram. Efficient approximation algorithms for scheduling malleable tasks. In *Proc. of the 11th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 23–32. ACM Press, 1999.
[102] A. Munier and C. Hanen. Using duplication for scheduling unitary tasks on m processors with unit communication delays. *Theoretical Computer Science*, 178(1–2):119–127, may 1997.
[103] R. Namyst and J.-F. Mehaut. PM^2: Parallel multithreaded machine. a computing environment for distributed architectures. In *Proc. of the International Conference of Parallel Computing*, 1995.
[104] V.Y. Pan. Solving polynomials with computers. *American Scientist—The Magazine of SIGMA XI, The Scientific Research Society*, 86:62–69, January–February 1998.
[105] C.H. Papadimitriou and J.D. Ullman. A communication-time tradeoff. *SIAM Journal on Computing*, 16(4):639–646, 1987.
[106] Christos H. Papadimitriou and Mihalis Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal of Computing*, 19(2):322–328, apr 1990.
[107] C. Picouleau. New complexity results on scheduling with small communication delays. *Discrete Applied Mathematics*, 60(1-3):331–342, 1995.
[108] R. Pollak. A hierarchical load balancing environment for parallel and distributed supercomputer. In *Proc. of the International Symposium on Parallel and Distributed Supercomputing (PDSC)*, 1995.
[109] R. Preis. *Analyses and Design of Efficient Graph Partitioning Methods.* PhD thesis, Universität Paderborn, 2000.
[110] R. Preis and R. Diekmann. The PARTY partitioning-library, user guide - version 1.1. Technical Report tr-rsfb-96-024, University of Paderborn, Sept. 1996.
[111] R. Preis and R. Diekmann. PARTY - A software library for graph partitioning. *Advances in Computational Mechanics with Parallel and Distributed Processing*, pages 63–71, 1997.
[112] V.J. Rayward-Smith. UET scheduling with unit interprocessor communication delays. *Discrete Applied Mathematics*, 18:55–71, 1987.
[113] G. Royle. Cages of higher valency. http://www.cs.uwa.edu.au/∼gordon/cages/allcages.html.
[114] S. Sahni and V. Thanvantri. Performance metrics: Keeping the focus on runtime. *IEEE Parallel and Distributed Technology*, 4(1):43–56, 1996.
[115] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, Cambridge, 1989.
[116] K. Schloegel, G. Karypis, and V. Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *Journal of Parallel and Distributed Computing*, 47(2):109–124, 1997.
[117] T. Schnekenburger and G. Stellner, editors. *Dynamic Load Distribution for Parallel Applications*, volume 24. Teubner, Teubner Texte zur Informatik edition, 1997.
[118] W. Schreiner, C. Mittermaier, and F. Winkler. On solving a problem in algebraic geometry by cluster computing. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Euro-Par 2000 Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 1196–1200. Springer-Verlag, 2000.
[119] U. Schwiegelsohn, W. Ludwig, J.L. Wolf, J. Turek, and P.S. Yu. Smart bounds for weighted response time scheduling. *SIAM Journal on Computing*, 28(1):237–253, 1999.
[120] W. Shu. Chare kernel – A runtime support system for parallel computations. *Journal of Parallel and Distributed Computing*, 11:198–211, 1991.
[121] M. Skutella and G.J. Woeginger. A PTAS for minimizing the weighted sum of job completion times on parallel machines. In *Proc. of the 31st ACM Symp. on Theory of Computing (STOC)*, pages 400–407. ACM, 1999.
[122] A. Steinberg. A strip-packing algorithm with absolute performance bound 2. *SIAM Journal on Computing*, 26(2):401–409, 1997.

[123] E. Tärnvik. Dynamo – a portable tool for dynamic load balancing on distributed memory multicomputers. *Practice and Experience*, 6(8):613–639, Dec. 1994.
[124] S. Tschöke. *Portable Parallel Branch-and-Bound Library: Reference Manual.* University of Paderborn, 1996. http://www.uni-paderborn.de/∼ppbb-lib.
[125] S. Tschöke, R. Lüling, and B. Monien. Eine Toolbox zur automatischen Parallelisierung beliebiger Branch & Bound Applikationen. In R. Grebe R. Flieger, editor, *Parallele Datenverarbeitung aktuell: TAT '94*, pages 149–154. IOS Press, 1994.
[126] S. Tschöke, R. Lüling, and B. Monien. Solving the traveling salesman problem with a distributed branch-and-bound algorithm. In *Proc. of the 9th International Parallel Processing Symposium (IPPS)*, pages 182–189. IEEE Computer Society Press, 1995.
[127] J. Turek, J. Wolf, and P. Yu. Approximate algorithms for scheduling parallelizable tasks. In *Proc. of the 4th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 323–332. ACM, 1992.
[128] L. G. Vailant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.
[129] J. Valdes, E.E. Tarjan, and E.L. Lawler. The recognition of series-parallel digraphs. *SIAM Journal of Computing*, 11:298–313, 1982.
[130] R.S. Varga. *Matrix Iterative Analysis.* Prentice-Hall, 1962.
[131] B. Veltman, B.J. Lageweg, and J.K. Lenstra. Multiprocessor scheduling with communication delays. *Parallel Computing*, 16:173–182, 1990.
[132] Q. Wang and K.H. Cheng. List scheduling of parallel tasks. *Information Processing Letters*, 37:291–297, 1991.
[133] Q. Wang and K.H. Cheng. A heuristic of scheduling parallel tasks and its analysis. *SIAM Journal on Computing*, 21(2):281–294, April 1992.
[134] J. B. Weissman and A. S. Grimshaw. A framework for partitioning parallel computations in hetergoneneous environments. *Concurrency – Practice & Experience*, 7(5):455–478, Aug. 1995.
[135] M. H. Willebeek-LeMair and A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, Sept. 1993.
[136] M.-Y. Wu. Symmetrical Hopping: A scalable scheduling algorithm for irregular problems. *Practice and Experience*, 7(7):707–736, Oct 1995.
[137] M.-Y. Wu and W. Shu. Scatter scheduling for problems with unpredictable structures. In *Proc. of the Distributed Memory Computing Conference*, pages 137–143, 1991.
[138] C. Xu and F.C.M. Lau. *Load Balancing in Parallel Computers.* Kluwer, 1997.
[139] T. Yang and C. Fu. Space / time-efficient scheduling and execution of parallel irregular computations. *ACM Transactions on Programming Languages and Systems*, 20(6):1195–1222, Nov. 1998.
[140] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994.
[141] T.-R. Yang and H.-X. Lin. Isoefficiency analysis of CGLS algorithms for parallel least squares problems. In B. Hertzberger and P. Sloot, editors, *Proc. of the International Conference of High-Performance Computing and Networking*, volume 1225 of *Lecture Notes in Computer Science*, pages 452–461. Springer, 1997.
[142] Christian Zimmermann. Efficient search in resource-based configuration: Parallelism and learning of heuristics. Master's thesis, Universität Paderborn, 2000.