Lastverteilungsalgorithmen für parallele Tiefensuche

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der Fakultät für Informatik der

Universität Karlsruhe

genehmigte

Dissertation

von

Peter Sanders

aus Ahlen

Tag der mündlichen Prüfung:	19. November 1996
Erster Gutachter:	Prof. Dr.-Ing. Roland Vollmar
Zweiter Gutachter:	Prof. Dr. rer. nat. Hartmut Schmeck

Vorwort

Bei meiner Diplomarbeit (SANDERS, 1993) ging es u.a. um die Parallelisierung eines Suchproblems aus der Automatentheorie. Dabei fiel mir auf, daß es zum Thema parallele Baumsuche zwar eine Vielzahl mehr oder weniger erfolgreicher praktischer Studien gab, daß aber ein Defizit an Algorithmenanalysen bestand, die geeignet wären, die Algorithmen unabhängig von Anwendung, Maschine und Implementierungsdetails zu bewerten und weiterzuentwickeln.

Ich bin Prof. Roland Vollmar dankbar, daß er es mir ermöglichte, dieser Frage im Rahmen meiner Promotion nachzugehen und dabei immer ein offenes Ohr für meine Probleme hatte. Mein Dank gilt auch Prof. Hartmut Schmeck für die Übernahme des Korreferats.

Sollte ich im Laufe der letzten Jahre etwas über wissenschaftliches Arbeiten gelernt haben, so ist dies zu einem großen Teil das Verdienst der „dienstälteren“ Kollegen am Lehrstuhl Informatik für Ingenieure und Naturwissenschaftler, Alf Christian Achilles, Henning Fernau, Heinrich Rust, Thomas Umland und Thomas Worsch.

Ohne eine Vielzahl von Diskussionen mit weiteren Bekannten, Kollegen und Studenten wäre diese Arbeit nicht denkbar. Auf die Gefahr hin, wichtige Personen zu vergessen, möchte ich dennoch einige Namen nennen: Markus Armbruster, Heiko Auerswald, Bernhard Beckert, Wolfgang Burke, Roger Buthenuth, Peter Diefenbach, Sebastian Egner, Sven Gilles, Stefan Hänssgen, Dominik Henrich, Ernst Heinz, Holger Hopp, Jochen Kaltenbach, Bernhard Klar, Thorsten Minkwitz, Markus Mock, Jörn Müller-Quade, Christian v. Roques, Susanne Wetzel, Wolf Zimmermann.

Durch das Internet, den Besuch von Konferenzen und einen Gastvortrag an der Universität Paderborn hatte ich außerdem die Möglichkeit, mich mit einer Reihe von Wissenschaftlern auszutauschen, die an ähnlichen Themen arbeiten. Auch hier erlaubt mir meine Vergeßlichkeit nur eine Auswahl von Personen zu nennen: A. Czumaj, T. Hagerup, L. Kale, R. Karp, T. Lauer, R. Lüling, F. Meyer auf der Heide, C. McDiarmid, B. Monien, A. Reinefeld, S. Tschöke und C. Xu.

Für den Zugang zu Parallelrechnern gilt mein Dank dem „Paderborn Center for Parallel Computing“ und dem Rechenzentrum der Universität Karlsruhe.

Inhaltsverzeichnis

Kapitel 1

Einführung

In dieser Arbeit werden Lastverteilungsalgorithmen untersucht, die eine effiziente Parallelisierung von Anwendungen ermöglichen, denen Tiefensuche in großen, implizit gegebenen Bäumen zugrunde liegt. Beispiele dafür sind Backtracking, Branch-and-bound durch Tiefensuche und Suche mit iterativem Vertiefen (z.B. KORF (1985)). Es werden existierende Lastverteilungsverfahren genauer als bisher analysiert und neue Algorithmen entwickelt. Dadurch wird der Stand der Technik bei der Parallelisierung von Tiefensuchverfahren soweit vorangetrieben, daß nun für eine große Klasse von Maschinen und Anwendungen bessere und zum Teil asymptotisch optimale Verfahren bekannt sind. Es handelt sich dabei um relativ einfache, randomisierte Algorithmen, deren Verhalten nicht nur im Mittel sondern auch für die schwierigsten Instanzen analytisch bestimmt werden kann.

Diese Einführung wird fortgesetzt, indem zunächst in Abschnitt 1.1 die Themenstellung motiviert wird. Dann gibt Abschnitt 1.2 einen Überblick über Struktur und Grundideen dieser Arbeit. Diese werden in Abschnitt 1.3 mit dem Stand der Technik in Beziehung gesetzt.

1.1 Motivation

Das hier betrachtete Thema läßt sich von zwei möglichen Sichtweisen her motivieren, die hier nebeneinandergestellt werden sollen: Der erste Ansatz geht vom allgemein großen Interesse an Leistungssteigerung durch Parallelverarbeitung aus, die durch die anhaltende Vergrößerung der Integrationsdichte logischer Schaltungen zunehmend attraktiv und praktikabel wird. Diese im Prinzip zur Verfügung stehende Rechenleistung läßt sich aber nur in höhere Leistung einer Anwendung ummünzen, wenn es gelingt, den in der Anwendung enthaltenen Parallelismus durch geeignete Algorithmen und Rechnerarchitekturen auszunutzen.

Manchmal ist die Angabe eines parallelen Algorithmus relativ leicht, weil oft schon die Beseitigung überflüssiger Reihenfolgefestlegungen in bekannten sequentiellen Algorithmen genügend Parallelität offenlegt. Ist der Rechenablauf darüberhinaus im voraus bekannt, hängt der Erfolg der Parallelisierung „nur“[1)] davon ab, ob und wie die Datentransportkapazität der zur Verfügung stehenden Maschine den Anforderungen gerecht werden kann. Für viele An-

[1)] Hinter diesem „nur“ verbergen sich leider oft NP-harte Planungsprobleme.

wendungen sind aber lediglich Algorithmen bekannt, deren Rechenablauf datenabhängig und oft sehr unstrukturiert ist. Dann ergibt sich das zusätzliche Problem, Daten und Berechnungen zur Laufzeit unter den Prozessoren aufzuteilen, ohne dabei Teile der Maschine derart zu überlasten, daß die Gesamtleistung beeinträchtigt wird. Außerdem sind konkrete Aussagen über die Systemleistung erst möglich, wenn das Zusammenwirken von Maschine, Lastverteilung und zugrundeliegender Anwendung berücksichtigt wird; die Analyse und der Entwurf von Lastverteilungsalgorithmen wird dadurch zu einem weiten und schwierigen Arbeitsfeld.

Deshalb ist es kaum zu vermeiden, die Untersuchung auf eine bestimmte Klasse von Anwendungen zu beschränken. Dies liefert den zweiten Ansatzpunkt für die Themenstellung. Viele Anwendungen im Operations Research und der Künstlichen Intelligenz beruhen auf der heuristischen Traversierung eines großen Zustandsraums. Die dabei entstehenden Suchräume sind oft so groß, daß nur eine implizite Aufzählung durch Tiefensuchtechniken wie Branch-and-bound, Backtracking und iteratives Vertiefen gangbar ist. Die Größe des Suchraums macht außerdem eine Parallelisierung der Suche erstrebenswert.[2)]

Eine Synthese der beiden Motivationsansätze ergibt sich daraus, daß die oben skizzierten Tiefensuchbäume eine sehr unregelmäßige Struktur haben können, auf der anderen Seite aber einen hohen Parallelitätsgrad und geringen Datenaustausch zwischen Teilproblemen aufweisen. Dadurch ergibt sich die Möglichkeit, Lastverteilung in einem Kontext zu untersuchen, bei dem davon auszugehen ist, daß die Güte des Verteilungsalgorithmus sich deutlich (vor allem auch positiv) auf die Effizienz des Gesamtsystems niederschlägt.

1.2 Überblick

Zunächst werden in Kapitel 2 einige Grundlagen für die Analyse randomisierter Algorithmen, den Entwurf paralleler Algorithmen und die Modellierung von Tiefensuchproblemen eingeführt.

Dieses Kapitel muß einer doppelten Anforderung gerecht werden. Einerseits sollen die Voraussetzungen für das Verständnis der Hauptergebnisse in möglichst kurzer und prägnanter Weise eingeführt werden. Andererseits erfordert die Komplexität der Materie eine weitergehende Diskussion und die Einführung zusätzlicher Konzepte. Diese Teile sind deshalb zur besseren Orientierung durch einen kleineren Schriftsatz kenntlich gemacht. Das gleiche gilt für weiterführende Diskussionen in den anderen Kapiteln.

In Kapitel 3 wird dann der in der Literatur oft recht unsystematisch und unübersichtlich behandelte Begriff des „asymptotischen Verhaltens mit hoher Wahrscheinlichkeit“ zu einem einfacher handhabbaren Mechanismus entwickelt, der sich so eng als möglich an den üblichen, deterministischen O-Kalkül anlehnt. Die Einführung von Hilfsmitteln wird abgeschlossen durch Kapitel 4, in dem einige auch anderweitig verwendbare Bausteine für Parallelalgorithmen beschrieben werden. Unter anderem werden einfache aber effiziente randomisierte Algorithmen für die Generierung von Zufallspermutationen und zur Beobachtung globaler Systemparameter wie Prozessorauslastung vorgestellt. In Kapitel 5 werden einige einfache

[2)] Der Gründlichkeit halber sei aber auch erwähnt, daß dieses Argument einen Pferdefuß hat. Wenn der Suchraum (wie nur allzuoft) exponenentiell mit der Problemgröße wächst, bedeutet selbst eine erfolgreiche Parallelisierung, daß sich nur um weniges größere Probleme als im sequentiellen Fall untersuchen lassen. Auch bei schnell wachsenden Suchräumen bleibt eine Parallelisierung aber interessant, wenn bei festgehaltener Problemgröße eine Beschleunigung der Suche oder eine „Verbesserung“ der Lösung gefragt ist. (Bei einigen Anwendungen im Operations Research sind im wahrsten Sinne des Wortes schon kleine Verbesserungen Gold wert.)

untere Schranken hergeleitet. Dadurch wird es möglich, einige der anschließend betrachteten Algorithmen als asymptotisch optimal einzustufen.

Der Hauptteil dieser Arbeit beginnt mit Kapitel 6, in dem zunächst der Ansatz des *zufälligen Anfragens* analysiert wird: Jeder Prozessor arbeitet an einem eigenen Teilproblem. Arbeitslos gewordene Prozessoren versuchen von zufällig ausgewählten anderen Prozessoren Arbeit zu bekommen, indem deren Teilprobleme aufgespalten werden. Wenn globale Kommunikation nicht deutlich teurer als lokale Kommmunikation ist, so ist dieser Algorithmus kaum zu übertreffen. Kapitel 7 zeigt, wie auf anderen Architekturen eine geeignete Mischung von globaler und lokaler Kommunikation zu noch besseren Ergebnissen führt.

Ist Kommunikation sehr teuer und der Suchraum nicht zu unregelmäßig, so kann die in Kapitel 8 beschriebene randomisierte statische Lastverteilung eingesetzt werden: Die Beschreibung des Gesamtproblems wird allen Prozessoren mitgeteilt und in eine große Zahl von Teilproblemen zerlegt, die zufällig auf die Prozessoren verteilt werden. Dies geschieht ohne weitere Kommunikation nur mit Hilfe der Prozessornummern und einem geeigneten Generator für Pseudozufallspermutationen. Das Verfahren läßt sich durch Kombination mit dynamischen Lastverteilungsalgorithmen weiter verbessern. Außerdem liefern die in Kapitel 8 angestellten Betrachtungen Einsichten in die Gründe der starken Unregelmäßigkeit von Suchbäumen.

Kapitel 9 faßt die Ergebnisse der Arbeit zusammen. In Anhang A wird über einige Implementierungserfahrungen berichtet. Am Ende der Arbeit finden sich außerdem ein kurzer Überblick über die verwendete mathematische Notation, ein Literaturverzeichnis und ein Stichwortverzeichnis.

1.3 Einordnung

Lastverteilung ist ein so weites Gebiet, daß ein Überblick, der allen möglichen Nuancen des Problems gerecht wird, kaum möglich ist. (So findet die Bibliographiedatenbank ACHILLES (1995a) allein 716 Literaturstellen, in denen der Begriff „load balancing“ vorkommt.) Deshalb soll nun zunächst anhand der Ziele dieser Arbeit eine grobe Abgrenzung vorgenommen werden.

Analytische Leistungsaussagen: Aufgrund der schon in der Motivation erwähnten komplexen Interaktionen zwischen Maschine, Lastverteilungsalgorithmus und Anwendung konzentrieren sich viele Arbeiten auf experimentelle Studien (HENRICH (1994); BECKER (1995)), während hier analytische Aussagen in den Vordergrund gestellt werden, die auch ohne eine Vielzahl von Simulationen, Implementierungen und Messungen Voraussagen über die Leistung eines Lastverteilungsalgorithmus für ein weites Spektrum von Anwendungsszenarien zulassen.

Skalierbarkeit: In dieser Arbeit soll es um Algorithmen gehen, die sich auch für Maschinen mit vielen Prozessoren kosteneffektiv einsetzen lassen. In diesem Kontext sind Verfahren, die zentralisierte Lastverteiler benutzen oder exzessiven Gebrauch von gemeinsamem Speicher machen, kaum geeignet.

Effizienz: Wenn irgend möglich, soll für hinreichend große Probleminstanzen eine Beschleunigung nahe der Prozessorzahl erreicht werden. Bei theoretischen Modellen, die jedem Pro-

zessor nur einen kleinen konstanten Speicherplatz zuweisen (z.B. KAKLAMANIS UND PERSIANO (1994); DEHNE ET AL. (1990a,b)), ist dies oft nicht möglich, da für jeden lokalen Berechnungsschritt komplexe Kommunikationsoperationen hinzukommen.

Nicht vorhersagbare Last: Eine fundamentale Eigenschaft von irregulären Baumsuchproblemen ist die Tatsache, daß im allgemeinen kaum vorhersagbar ist, welcher Rechenaufwand sich hinter einem Teilproblem verbirgt. Oft wäre eine hinreichend genaue Bestimmung dieser Größe ähnlich aufwendig wie eine komplette *Lösung* des Teilproblems. Viele Verfahren aus den Bereichen Scheduling (z.B. YU UND GHOSAL (1992)), Teile-und-herrsche (divide-and-conquer) (z.B. MAYR UND WERCHNER (1993)) und die meisten Offline-Algorithmen (z.B. MEYER AUF DER HEIDE ET AL. (1993)) setzen die Teilproblemgröße aber als bekannt voraus und sind deshalb für Baumsuche im allgemeinen nicht einsetzbar.

Problembezogene Leistungsaussagen: Viele Arbeiten beruhen auf abstrakten Lastverteilungsmodellen, bei denen es schwierig ist, Aussagen über Geschwindigkeit und Qualität der Lastverteilung in Voraussagen über die Leistung des Gesamtsystems Maschine, Lastverteiler *und* Anwendung umzusetzen. Ein Beispiel sind Varianten des *Token distribution problem* (PELEG UND UPFAL, 1989), bei dem es um möglichst schnelles, (mehr oder weniger) gleichmäßiges Verteilen unabhängiger Lastpakete geht (MEYER AUF DER HEIDE ET AL. (1993); LAUER (1995); GHOSH ET AL. (1995); SANDERS UND WORSCH (1995); SANDERS (1996b), u. v. a. m.) Für viele Anwendungen ist aber unklar, wann wo wieviele Lastpakete wie schnell zu verteilen sind. Dies wäre aber nötig, um von der abstrakt definierten Qualität eines Verteilungsalgorithmus auf die Leistung des Gesamtsystems zu schließen.

Eignung für Tiefensuche: Schließlich sei noch erwähnt, daß selbst Lastverteilungsalgorithmen, die für scheinbar verwandte Gebiete wie parallele Bestensuche[3)] entworfen wurden, für Tiefensuche nicht immer gut geeignet sind. Diese Verfahren basieren z.B. oft auf der Identifizierung von Suchbaumknoten mit zu balancierenden Lastpaketen (z.B. VORNBERGER (1987); FELTEN (1988); LÜLING ET AL. (1991); LÜLING UND MONIEN (1992); MCKEOWN ET AL. (1992); KARP UND ZHANG (1993); HENRICH (1994)). Für Bestensuche ist das sinnvoll, da dadurch die Knotenbewertung für den Lastverteiler sichtbar gemacht wird. Bei Tiefensuche ist dies jedoch unnötig. Die hier beschriebenen Algorithmen benutzen den bei Tiefensuche ohnehin benötigten Stapel zur kompakten Repräsentation ganzer Teilbäume, die nur bei Bedarf weiter unterteilt werden.

Trotz der vielfältigen Abgrenzungsmöglichkeiten steht diese Arbeit natürlich nicht isoliert im Raum. Der grundlegende Algorithmus für zufälliges Anfragen, auf dem Kapitel 6 aufbaut, ist wohl von mehreren Gruppen unabhängig voneinander entdeckt worden (FINKEL UND MANBER, 1987; KUMAR UND RAO, 1990; MONIEN ET AL., 1990) und hat sich auch schon für ein breites Spektrum von Anwendungen bewährt (ARVINDAM ET AL., 1990b,a; KUMAR UND ANANTH, 1991; KUMAR ET AL., 1994; FELDMANN, 1993; FELDMANN ET AL., 1991, 1994). In KUMAR UND ANANTH (1991) wird zufälliges Anfragen empirisch als den meisten deterministischen Verfahren (z.B. MA ET AL. (1988); RAO UND KUMAR (1987a,b); LIN (1991); KERGOMMEAUX UND CODOGNET (1994); NAGANUMA (1993)) überlegen nachgewiesen. Die vorliegende Arbeit begründet die Überlegenheit von zufälligem Anfragen

[3)] Also Baumtraversierungsverfahren, bei denen Baumknoten eine Bewertung haben und jeweils der beste noch nicht betrachtete Knoten als nächstes in Angriff genommen wird.

analytisch (siehe auch SANDERS (1994a,b)) und zeigt in Anhang A empirisch, daß zufälliges Anfragen selbst für große Maschinen mit hoher Latenzzeit gut geeignet ist (siehe auch SANDERS (1994e, 1995d, 1996d)). Auf solchen Maschinen sind allerdings weitere (auch asymptotische) Verbesserungen möglich (Kapitel 7 sowie SANDERS (1995a,e)). Die so entstandenen Algorithmen lassen sich auch als Verbesserungen der teilweise wenig praktikablen Baumeinbettungsalgorithmen aus LEIGHTON ET AL. (1989) und RANADE (1994) auffassen. Kürzlich ist zufälliges Anfragen in BLUMOFE UND LEISERSON (1994); BLUMOFE ET AL. (1995) dahingehend weiterentwickelt worden, daß nun auch bestimmte Abhängigkeiten zwischen Teilproblemen effizient behandelt werden können.[4)]

Der in Kapitel 8 betrachtete Algorithmus basiert auf noch älteren Ansätzen. Die Idee, ein Wurzelproblem nach einem Broadcast aufgrund der Prozessornummer – und ohne weitere Kommunikation – in disjunkte Teilprobleme zu zerteilen, geht auf EL-DESSOUKI UND HUEN (1980) zurück und wird heute oft für die Initialisierung paralleler Baumsuchverfahren eingesetzt (MA ET AL., 1988; LÜLING UND MONIEN, 1992; SANDERS, 1993, 1994a; HENRICH, 1994). Weite Verbreitung gefunden hat die statische Lastverteilung durch *Scatter decomposition* (NICOL UND SALTZ, 1990; KUMAR ET AL., 1994) oder zufällige Plazierung einer hinreichend großen Zahl von Teilproblemen (KRUSKAL UND WEISS, 1985). Die dort erzielten Ergebnisse lassen sich jedoch nicht direkt auf Baumsuche anwenden. Die Kombination dieser beiden Ideen (vergleiche auch SANDERS (1994f, 1996a)) ist vorher noch nicht untersucht worden. Statt dessen gibt es eine Reihe von Arbeiten über *randomisierte Suche*, bei der die Prozessoren unabhängig voneinander zufällige Zweige des Suchbaums verfolgen (JANAKIRAM ET AL., 1987, 1988; NATARAJAN, 1989; ERTEL, 1992; SANDERS, 1993). Dies ist allerdings nur in speziellen Fällen sinnvoll, da der Suchraum nicht oder nur sehr langsam vollständig ausgeschöpft wird.

[4)] Die bisher realisierten Anwendungen lassen sich aber alle als Baumsuchverfahren auffassen.

Kapitel 2

Modellierung und Grundlagen

Dieses Kapitel dient dazu, die Arbeit auf eine tragfähige Grundlage zu stellen. Zunächst werden in Abschnitt 2.1 einige Begriffe erklärt, die für die Beschreibung und Analyse randomisierter Algorithmen benötigt werden. Es folgen mit Abschnitt 2.2 einige Grundlagen aus der Parallelverarbeitung. Schließlich führt Abschnitt 2.3 den Begriff der *baumförmigen Berechnung* als Modell für parallele Tiefensuche ein und setzt es mit anderen Modellen und tatsächlichen Anwendungen in Beziehung. Aufbauend auf der so komplettierten Begriffsbildung werden in Abschnitt 2.4 einige grundlegende Ansätze zur Parallelisierung baumförmiger Berechnungen vorgestellt. Das Kapitel wird in Abschnitt 2.5 zusammengefaßt.

2.1 Randomisierte Algorithmen

Randomisierte Algorithmen unterscheiden sich nur dadurch von deterministischen Algorithmen, daß Zufallsgeneratoren eingesetzt werden. Eine weitere Klassifizierung ergibt sich nach Art und Zweck des Einsatzes von Zufallsgeneratoren. Alle hier betrachteten randomisierten Algorithmen sind „Las Vegas Algorithmen“ im Sinne von MOTWANI UND RAGHAVAN (1995). Diese terminieren immer, und die Qualität des Ergebnisses ist genauso gut oder schlecht wie die eines vergleichbaren deterministischen Algorithmus. Lediglich die Ausführungszeit hängt von den getroffenen Zufallsentscheidungen ab. Für jede einzelne Probleminstanz[1)] P ist die Ausführungszeit $T(P)$ eine Zufallsvariable. Der der Analyse zugrundeliegende Ereignisraum besteht aus den vom Algorithmus verwendeten Zufallsobjekten. Dies können z.B. Zufallszahlen, Zufallsbits („Münzwürfe“), Zufallspermutationen oder Folgen und Tupel einfacherer Zufallsobjekte sein. Wenn nicht ausdrücklich anders festgelegt, sind alle Elemente des Ereignisraums gleichwahrscheinlich.

Entscheidend für die Bewertung von deterministischen und randomisierten Algorithmen ist im folgenden das Verhalten für die *schwierigsten* Probleminstanzen. Dies ist ein wichtiger Unterschied zur Analyse des Verhaltens *im Mittel* bei der aus Annahmen über die Auftretenswahrscheinlichkeiten von Probleminstanzen Aussagen über die Ausführungszeit für „typische“ Probleminstanzen hergeleitet werden.

[1)] Wenn dies aus dem Kontext ersichtlich ist, wird im folgenden statt „Probleminstanz“ auch kurz „Problem“ benutzt.

Ein naheliegender Ansatz zur Beschreibung der Ausführungszeit ist es, ihren Erwartungswert $\mathbf{E}T(P)$ zu berechnen. Eine günstige erwartete Ausführungszeit bedeutet allerdings noch nicht, daß keine großen Abweichungen in der Ausführungszeit auftreten können, die u.U. die Nützlichkeit des randomisierten Algorithmus einschränken.

Durch zusätzliche Betrachtungen können z.B. die Varianz oder andere aus der Wahrscheinlichkeitstheorie bekannte Maßzahlen berechnet werden. Ein Grund, die Varianz als interessante Maßzahl anzusehen, wird in GRAHAM ET AL. (1992) angeführt:

> "But we haven't really seen a good reason why the variance is a natural thing to compute. Everybody does it, but why? The main reason is *Cheybyshev's inequality* [...], which states that the variance has a significant property:
>
> $$\mathbf{P}\left[(X-\mathbf{E}X)^2 \geq \alpha\right] \leq \frac{\mathbf{Var}X}{\alpha} \quad .$$"

Die Varianz kann also genutzt werden, um Schranken für die Wahrscheinlichkeit großer Ausführungszeiten anzugeben. Oft lassen sich jedoch viel schärfere Schranken ohne den (oft mühsamen) Umweg über die Varianz angeben. Aus diesem Ansatz ergeben sich Aussagen der Form $\mathbf{P}[T(P) \leq f(P)] \geq 1 - g(P)$ bzw. $\mathbf{P}[T(P) > f(P)] < g(P)$, wobei g eine Funktion ist, die für große P gegen Null geht. Aussagen dieser Art sind heute bei der Beschreibung randomisierter paralleler Algorithmen allgemein üblich. Die große Flexibilität dieses Ansatzes hat allerdings den Nachteil, daß von Autor zu Autor oder gar von Aufsatz zu Aufsatz unterschiedliche Konventionen zugrunde gelegt werden. Dies macht die Ergebnisse schwer vergleichbar und schwer nachvollziehbar. Außerdem müssen viele Rechnungen wieder und wieder durchgeführt werden, da die Vielzahl verwendeter Varianten die Formulierung wiederverwendbarer Lemmata erschwert.

Die bevorzugte Analysemethode in dieser Arbeit beruht auf *asymptotischen Verhalten mit hoher Wahrscheinlichkeit* und lehnt sich stark an den deterministischen O-Kalkül an. Die in der folgenden Definition beschriebene Variante hat sich für parallele Algorithmen als aussagekräftig und gut handhabbar bewährt.

Definition 2.1. Sei $f \in {\mathbb{R}_*}^{\mathbb{N}}$ und X eine Zufallsvariable mit $\mathbf{Bild}\, X \subseteq \mathbb{R}_*$, die von einem Parameter n abhängen kann. Dann ist X *mit hoher Wahrscheinlichkeit*[2)] in $\mathrm{O}(f(n))$ oder kurz $X \in \tilde{\mathrm{O}}(f(n))$ gdw.

$$\forall \beta \in \mathbb{R}_+ : \exists c \in \mathbb{R}_+ : \exists n_0 : \forall n \geq n_0 : \mathbf{P}[X \leq cf(n)] \geq 1 - n^{-\beta}.$$

Man sagt auch „Die Wahrscheinlichkeit von $X > cf(n)$ ist *polynomiell klein*".

Es lassen sich analoge Definitionen für Funktionen mit mehreren Veränderlichen angeben. Die Wahrscheinlichkeitsaussage dieses $\tilde{\mathrm{O}}$-*Kalküls* bezieht sich dabei aber weiter auf einen einzigen ausgezeichneten Parameter. Hier ist dies immer n und damit meist die Prozessorzahl[3)].

Definition 2.1 orientiert sich an LEIGHTON ET AL. (1989, 1994), IERARDI (1994) und anderen. Mit der Reihenfolge der Quantoren wird es allerdings im allgemeinen nicht sehr genau genommen und n_0 fällt meist völlig unter den Tisch. Etwas stärker ist die in RAJASEKARAN (1992) und REIF UND SEN (1994) verwendete Definition:

$$\exists c \in \mathbb{R}_+ : \forall \beta \in (\mathbb{R}_+ \setminus [0,1]) : \forall n : \mathbf{P}[X \leq c\beta f(n)] \geq 1 - n^{-\beta}.$$

[2)] Man beachte die Ähnlichkeit zu der Eigenschaft $\exists c \in \mathbb{R}_+ : \exists n_0 : \forall n \geq n_0 : \mathbf{P}[X \leq cf(n)] = 1$, die gilt, wenn X mit Sicherheit in $\mathrm{O}(f(n))$ ist.

[3)] Diese Konvention ist vor allem nützlich, wenn die Skalierbarkeit paralleler Algorithmen untersucht werden soll. Dann werden nämlich oft alle Maschinen- und Problemparameter als Funktionen von n aufgefaßt.

Die Konstante vor f darf also nur linear mit dem Exponenten β steigen. Für eine Vervollständigung zwecks Ausschluß kleiner n wären denkbar:

$$\exists c \in \mathbb{R}_+ : \forall \beta \in \mathbb{R}_+ \geq 1 : \exists n_0 : \forall n \geq n_0 : \ldots$$

$$\exists c \in \mathbb{R}_+ : \exists n_0 : \forall \beta \in \mathbb{R}_+ \geq 1 : \forall n \geq n_0 : \ldots$$

oder gar

$$\exists c \in \mathbb{R}_+ : \exists n_0 : \forall \beta \in \mathbb{R}_+ \geq 1 : \forall n \geq \beta n_0 : \ldots$$

Wegen ihrer Kompliziertheit werden diese Varianten hier nicht weiter verwendet. Die meisten Ergebnisse lassen sich jedoch geeignet modifizieren. Schließlich wird der Begriff *mit* sehr *hoher Wahrscheinlichkeit* (with very high probability) in LEIGHTON (1992) mit Wahrscheinlichkeiten in $1 - O\left(2^{-n^\varepsilon}\right)$ gleichgesetzt.

Deutlich schwächere Definitionen des Begriffs *mit hoher Wahrscheinlichkeit* finden sich z.B. ebenfalls in LEIGHTON (1992). Dort gelten Wahrscheinlichkeiten in $1 - O(1/n)$ schon als groß. Die Wahl eines festen Exponenten für $1/n$ hat den Nachteil, daß die in Kapitel 3 eingeführten Schlußregeln wie „$X \in \tilde{O}(f) \wedge Y \in \tilde{O}(f) \implies X + Y \in \tilde{O}(f+g)$“ nicht mehr gelten. Gelegentlich finden sich Algorithmen, bei denen der Exponent zwar beliebig groß gemacht werden kann, aber von im Programm verwendeten Konstanten abhängt. Bei den hier beschriebenen Algorithmen wird dies wenn möglich vermieden. Tritt dieser Fall trotzdem einmal auf, so wird dies ausdrücklich erwähnt. (Ein Beispiel findet sich in Abschnitt 4.2.)

2.2 Parallelrechner

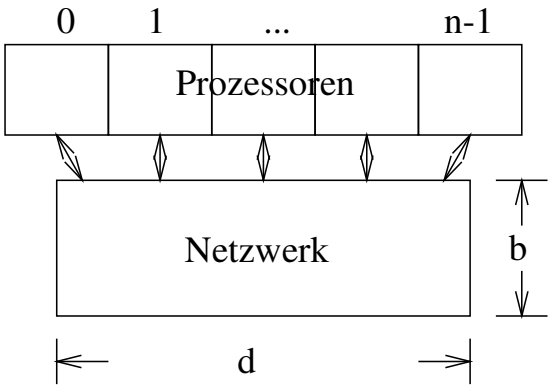

Abbildung 2.1: Grundlegendes Maschinenmodell.

Ziel dieses Abschnittes ist es, ein Maschinenmodell einzuführen, das einerseits einfach ist, andererseits den Eigenheiten möglichst vieler wichtiger Architekturen gerecht wird. Betrachtet werden Parallelrechner, die aus n identischen durch ein Verbindungsnetzwerk verbundenen Prozessoren (PEs = Prozessorelementen) bestehen. Abbildung 2.1 zeigt dies in einer schematischen Darstellung. Die PEs sind von 0 bis $n-1$ durchnumeriert und haben eine Wortlänge von $\Omega(\log n)$ Bits. Arithmetische und logische Operationen sowie das Erzeugen von hinreichend guten unabhängigen Pseudozufallszahlen mit $O(\log n)$ Bits können in konstanter Zeit bewerkstelligt werden. (In LEIGHTON (1992) wird dies als das *word model* bezeichnet.) Es

wird kein gemeinsamer Speicher vorausgesetzt.[4)] Wenn nicht besonders vermerkt, arbeiten die PEs jeweils mit einem eigenen Befehlsstrom (MIMD, Multiple Instruction Multiple Data).

Die meisten der Algorithmen werden (zunächst) als synchrone Algorithmen formuliert, d.h., die Berechnung wird in Rechen- und Kommunikationsphasen eingeteilt, an denen jeweils alle PEs teilnehmen. Solche Algorithmen sind meist einfacher zu verstehen und zu analysieren. Die dadurch implizierten Synchronisationen verursachen auf SIMD-Rechnern (Single Instruction Multiple Data) und auf MIMD-Rechnern, die über Synchronisationshardware oder eine global synchronisierte Uhr verfügen, keine signifikanten Kosten. Auch bei anderen Maschinen tritt der Synchronisationsaufwand meist gegenüber den Kommunikationskosten in den Hintergrund. In Fällen, wo dies anders ist (z.B. Abschnitt 7.3), wird im einzelnen darauf eingegangen, wie die Synchronisationskosten vermieden werden können.

Ein wichtiges Schwestergebiet zur Parallelverarbeitung sind die verteilten Systeme. Es werden hier zwar einige der dort entwickelten Verfahren „ausgeliehen“ (z.B. die in Abschnitt 4.1 beschriebene Terminierungserkennung), es gibt aber eine deutliche Abgrenzung bezüglich des Maschinenmodells. So spielen Fehlertoleranz oder Sicherheitsaspekte im folgenden keine Rolle. Nachrichten werden innerhalb definierter Zeitschranken ausgeliefert und verursachen nur insoweit Kosten als sie zur parallelen Ausführungszeit beitragen. Sowohl PEs als auch Netzwerk sind ganz der Bearbeitung der Anwendung gewidmet.

Parallele Algorithmen werden hier meist durch Angabe eines sequentiellen Programms formuliert, das auf allen PEs parallel mit jeweils eigenem Namens- und Adreßraum abläuft. Ein entfernter Zugriff auf eine Variable v von PE i wird durch $v\langle i\rangle$ dargestellt. Für diesen Zugriff fallen die Kosten eines Nachrichtenaustauschs an. Die lokale Prozessornummer ist in der vordefinierten Variablen i_{PE} gespeichert.

Im folgenden werden zunächst in Abschnitt 2.2.1 einige der wichtigeren Verbindungsnetzwerke beschrieben. Die Lastverteilungsalgorithmen werden allerdings weitgehend in netzwerkunabhängiger Form formuliert und basieren deshalb auf in den Abschnitten 2.2.2 und 2.2.3 beschriebenen portablen Operationen zum Verschicken von Nachrichtenpaketen und für kollektive Operationen wie globale Summenbildung. Neben dem praktischen Vorteil der Portabilität vereinfacht sich dadurch auch die Algorithmenanalyse, die modular aus der Analyse des portablen Algorithmus und der Analyse der Basisalgorithmen zusammengesetzt werden kann. Schließlich werden einige Maßzahlen diskutiert, die der Charakterisierung der Leistung paralleler Algorithmen dienen.

2.2.1 Verbindungsnetzwerke

Im allgemeinen Fall kann das Verbindungsnetzwerk eines Parallelrechners ein kompliziertes Gebilde aus PEs, Speichern, Schaltern und Verbindungen zwischen ihnen sein. Mathematisch läßt sich dies am ehesten durch einen Hypergraphen G erfassen. Unter der Vielzahl möglicher Maßzahlen zur Charakterisierung von Verbindungsnetzwerken (siehe auch HWANG (1993)) spielen die folgenden beiden eine besonders wichtige Rolle: Der *Durchmesser* d von G ist die maximale Entfernung zwischen zwei PEs gemessen in der Anzahl dazwischenliegender Kanten. Der Durchmesser ist (asymptotisch) entscheidend für die Verzögerung, die eine zwischen weit entfernten PEs auszutauschende Nachricht in einem niedrig belasteten Verbindungsnetzwerk erfährt. Bei hohem Nachrichtenverkehr ist dagegen die *Bisektionsbreite* b als Maß für die

[4)] Aus den hier entwickelten Algorithmen lassen sich effiziente Implementierungen für Rechner mit gemeinsamem Speicher ableiten, die durch hohe Lokalität und seltene Speicherzugriffskonflikte charakterisiert sind.

Gesamtbandbreite bei globaler Kommunikation mindestens genauso wichtig. b ist definiert als die minimale Zahl von Kanten, die es zu entfernen gilt, um G in zwei Zusammenhangskomponenten zu teilen, die $\lceil n/2 \rceil$ bzw. $\lfloor n/2 \rfloor$ PEs enthalten.

Da es sehr schwierig ist, hinreichend genaue Aussagen zu machen, die für alle denkbaren Verbindungsnetzwerke gelten, betrachtet diese Arbeit nur eine Auswahl wichtiger Verbindungsnetzwerke. Wie die Auflistung der realen Maschinen, die diese Architekturen verwenden, zeigt, ist damit ein großer Teil der relevanten Architekturen abgedeckt. Da später nur an wenigen Stellen von den Netzwerkeigenschaften Gebrauch gemacht wird, lassen sich zusätzliche Topologien relativ leicht nachträglich einbauen.

Vollständige Verknüpfung wird meist durch Kreuzschienenverteiler realisiert. Es gilt $d \in \Theta(1)$ und $b \in \Omega(n)$.

Die Annahme der vollständigen Verknüpfung ist oft eine Abstraktion der unten beschriebenen mehrstufigen Netze. Dies ist für realistische n gerechtfertigt, wenn die beteiligten Schalter eine im Vergleich zur Nachrichtenaufsetzzeit vernachlässigbare Verzögerung bewirken. Dieser Schluß läßt sich aber nicht für Netzwerke mit $b \notin \Omega(n)$ (wie z.B. Gitter) verallgemeinern, weil diese begrenzte Bisektionsbreite bei starkem Nachrichtenverkehr einen Engpaß darstellt.

Mehrstufige Netze: Es gibt eine Vielzahl von Netzwerken, die gemeinsam haben, daß sie $\Theta(n \log n)$ Schalter enthalten und ein Netz mit $d \in \Theta(\log n)$ und $b \in \Theta(n)$ realisieren. *Fat trees* (LEISERSON, 1985) sind hierarchische mehrstufige Verbindungsnetzwerke, bei denen Nachrichten auf einer einzigen Netzschicht sowohl injiziert als auch wieder abgeholt werden.[5)] Fat trees lassen sich in Teilnetzwerke der Größe 2^k aufteilen, innerhalb derer in $O(k)$ Schritten kommuniziert werden kann.

Allein in LEIGHTON (1992) werden die folgenden Netzwerktypen genannt: *Butterfly-*, *Beneš-*, *Omega-*, *(reverse) Baseline-*, *Banyan-* und *Delta-Netzwerk*. Andere häufiger erwähnte Varianten sind *CLOS*-Netzwerke (FUNKE ET AL., 1992) und verschiedene Expandergraphen (z.B. CHONG ET AL. (1994)). Beispiele für reale Maschinen mit mehrstufigem Verbindungsnetzwerk sind die Thinking Machines CM-5 (HILLIS UND TUCKER, 1993), die Meiko CS-2 (MEIKO, 1993a,b), die MasPar MP-1 und MP-2 (PRECHELT, 1993a), die IBM SP (SNIR ET AL., 1995) und der Parsytec SuperCluster (FUNKE ET AL., 1992)[6)] und Cognitive Computer.

Hyperwürfel: Ist $n = 2^k$, dann ist ein k-dimensionaler Hyperwürfel (Hypercube) ein Verbindungsnetzwerk, bei dem PE i mit den PEs aus der Menge $\left\{ i \oplus 2^j \mid j \in \mathbb{N}_k \right\}$ verbunden ist. Es gilt $d = \log n$ und $b = n/2$.

Diese hohe Bisektionsbreite kann allerdings nur dann voll ausgenutzt werden, wenn ein Knoten in der Lage ist, $\log n$ Leitungen gleichzeitig zu bedienen. Während Hyperwürfel in den 80er Jahren eine der am weitesten verbreiteten Parallelrechnerarchitekturen waren (z.B. HWANG (1993)), hält gegenwärtig nur noch NCube an dieser Architektur fest. Wichtige Gründe dafür dürften die begrenzte Skalierbarkeit und die hohen Kosten für $n \log n$ recht lange und schwer ausnutzbare Verbindungen sein. Wegen seiner einfachen Struktur ist der Hyperwürfel aber immer noch ein wichtiges Modell zum Entwurf von Algorithmen, die dann durch Emulation auf einem anderen Verbindungsnetzwerk ablaufen können.

Hyperwürfelartige Netze sind solche Verbindungsnetzwerke, auf denen *normale Hyperwürfelalgorithmen* im Sinne von LEIGHTON (1992) effizient ausgeführt werden können. Bei diesen Algorithmen darf nur in einer Würfeldimension i gleichzeitig kommuniziert werden und

[5)] In LEISERSON (1985) werden fat trees anders eingeführt. Die hier verwendete Darstellung macht aber die Verwandtschaft zu mehrstufigen Netzwerken besser deutlich.

[6)] Die gewählte Verbindungsstruktur ist hier allerdings zur Programmlaufzeit nicht mehr änderbar.

dies auch nur, wenn im vorherigen Zeitschritt[7)] entweder entlang Dimension $i-1 \bmod \log n$, i oder $i+1 \bmod \log n$ kommuniziert wurde.

Aus vielen der oben genannten mehrstufigen Netzwerke ergeben sich hyperwürfelartige Netze, wenn PEs nicht nur in der untersten Schicht sondern an jedem Knoten des Verbindungsgraphen existieren. Außerdem sind *Cube-connected-cycle-*, *Shuffle-exchange-* und *de Bruijn-Netzwerke* zu nennen. Wie Hyperwürfel haben hyperwürfelartige Netze logarithmischen Durchmesser und kommen dabei mit konstantem Knotengrad aus. Mit $\Theta(n/\log n)$ ist die Bisektionsbreite etwas kleiner. Das de Bruijn-Netz wird im Parallelerechner *Triton* (PHILIPPSEN ET AL., 1992) verwendet und eignet sich wegen seines Knotengrades von vier auch gut zur Verschaltung von Transputern (z.B. FELDMANN (1993)).

Gitter: Ein r-dimensionales Gitter ergibt sich, wenn den PEs Koordinaten aus einem quaderförmigen Bereich aus $\mathbb{N}^r$ zugeordnet werden können, so daß genau solche PEs verbunden sind, deren euklidischer Abstand eins ist. Zur Vermeidung unproduktiver Notation wird davon ausgegangen, daß $n^{1/r} \in \mathbb{N}$ und daß die PEs einen Würfel bilden. Dann ergibt sich $d = rn^{1/r} - r$ und $b = n^{1-1/r}$. Sind zusätzlich PEs am Rand des Gitters mit den ihnen gegenüberliegenden PEs verbunden, so ergeben sich *Ring-* ($r = 1$) bzw. *Torusnetzwerke*.

Gitter zeichnen sich vor allem durch ihre Einfachheit aus. Ein 2D-Gitter kommt mit $O(n)$ Chipfläche aus, während die vorher beschriebenen Netzwerke durchweg $\Omega(n^2)$ bzw. $\Omega(n^2/\log n)$ Fläche benötigen (ULLMAN, 1984). Außerdem lassen sich Gitter für $r \leq 3$ mit Verbindungen konstanter Länge aufbauen, so daß die Annahme konstanter Signalverzögerungszeiten tatsächlich realistisch ist. Schließlich sei erwähnt, daß für r-dimensionale Gitter entworfene Algorithmen auf r-dimensionalen *Meshes of trees* emuliert werden können (ACHILLES, 1995b). Beginnend beim ersten parallelen Superrechner überhaupt, dem Illiac IV (VOLLMAR UND WORSCH, 1995), sind eine Vielzahl von Rechnern mit Gittern aufgebaut worden. Neuere Beispiele sind die Rechner der Firma MasPar, die GC-Baureihe der Firma Parsytec, und die Intel Paragon, die alle 2D-Gitter enthalten. Die Cray T3D/T3E (OED, 1993) basiert auf einem 3D-Gitter.

Busse: Eine naheliegende Art, ein paralleles System zu erhalten, besteht darin, mehrere Prozessoren an einem einzigen Bus zu betreiben. Die PEs werden üblicherweise mit einem lokalen Speicher ausgestattet. Es gilt $d = 1$ und $b = 1$.

Die begrenzte Bandbreite macht das Erreichen einer guten Effizienz für große n recht schwierig. Eine große Rolle spielen lokale Netze auf der Basis von Bussen wie das *Ethernet* und seine Nachfolger. Trotz ihrer niedrigen Kommunikationsleistung sind lokale Netze attraktiv für die Parallelverarbeitung, da unbeschäftigte vernetzte Arbeitsplatzrechner fast überall vorhanden sind.

Hybride Netzwerke: Durch Kombination verschiedener Netzwerkarchitekturen können Rechner effizienter bzw. billiger gemacht werden. Eine Möglichkeit ist die Leistungserhöhung durch gleichzeitiges Anbieten mehrerer Netzwerke: Die Rechner der Firma MasPar enthalten eine Kombination aus 2D-Gitter, mehrstufigem Netzwerk und Bus. Die Thinking Machines CM-5 kombiniert einen Fat tree mit einem Baumnetzwerk zum schnellen Berechnen von globalen Summen u.ä. Eine andere Möglichkeit ist eine hierarchische Anordnung. Eine typische Struktur eines lokalen Netzes sieht heute z.B. oft so aus, daß einige Workstations an einem gemeinsamen Ethernetstrang hängen, mehrere Stränge werden dann über ein leistungsfähiges Schaltelement verbunden, um die verfügbare Bandbreite zu erhöhen. Nach unten ist die Hierarchie oft dadurch fortgesetzt, daß eine einzige Workstation mehrere PEs an einem Bus betreibt (z.B. Sun SS10/20). Nach oben können z.B. mehrere lokale Netze zu einem Campus-Netzwerk (z.B. über FDDI) verbunden sein, das seinerseits im Internet hängt. Im allgemeinen gilt dabei, daß die pro PE nutzbare Kommunikationsbandbreite von unten nach oben abnimmt.

[7)] Für diese Definition wird ein synchron arbeitender Rechner angenommen.

2.2.2 Datentransportprobleme

Grundbaustein jeder Interaktion zwischen PEs ist im vorliegenden Maschinenmodell der Transport von Nachrichtenpaketen konstanter Länge (gemessen in Maschinenworten) von einem PE A zu einem PE B. Ein Paket kann in konstanter Zeit zu einem im Verbindungsnetzwerk benachbarten PE (oder Schaltelement) transportiert werden. Eine Verbindung kann zu einem gegebenen Zeitpunkt immer nur für den Transport eines einzigen Pakets verwendet werden.

Wie lange die Auslieferung eines Pakets dauert, wenn A und B nicht benachbart sind, hängt stark vom verwendeten Datentransportalgorithmus (Routing-Algorithmus) und dem vorliegenden Kommunikationsmuster ab. Die sich daraus ergebende Problematik ist für sich schon ein weites Forschungsgebiet. Um nicht von der Themenstellung dieser Arbeit abzuschweifen, wird deshalb versucht, weitestgehend auf bekannte Algorithmen und Ergebnisse zurückzugreifen; auch wenn viele der existierenden Beweise streng genommen neu aufgerollt werden müßten, um sie an eine bestimmte Kombination von Verbindungsnetzwerk und Datentransportalgorithmus oder die hier verwendete Definition des Begriffs „mit hoher Wahrscheinlichkeit“ anzupassen.

Die Beweise der folgenden grundlegenden Aussagen lassen sich zum großen Teil in LEIGHTON (1992) nachlesen. Wenn nicht anders gesagt, gelten sie für alle oben beschriebenen Netzwerktypen und lassen sich mit Hilfe von Verfahren realisieren, die keinen übermäßigen Speicherbedarf für die Pufferung weiterzuleitender Pakete haben. Die Aussagen lassen sich sicher *nicht* auf beliebige Netze übertragen. Dies ist aber auch nicht nötig, da Netzwerktypen mit anderen Eigenschaften in der Regel für die Konstruktion von Parallelrechnern wenig geeignet sind.

Satz 2.2. *In einem sonst unbelasteten Netz benötigt der Austausch von k Paketen zwischen zwei beliebigen Prozessoren Zeit $\Theta(d+k)$.*

Um diese Leistung zu erreichen, wird *Pipelining* verwendet, d.h., in jedem Zeitschritt wird ein neues Paket ins Netz „gepumpt“ und alle im Netz befindlichen Pakete, die ihr Ziel noch nicht erreicht haben, werden zum nächsten PE oder Schaltelement transportiert.

Entsprechende Aussagen gelten, wenn mehrere gleichzeitig stattfindende Nachrichtenaustauschvorgänge sich nicht gegenseitig stören. Außerdem läßt sich die Aussage auf ein Teilnetz mit geringerem Durchmesser beziehen, solange es beide Kommunikationspartner umfaßt. Sinngemäß gelten diese Verallgemeinerungen auch für die folgenden Aussagen.

Definition 2.3. Eine k-Permutation ist ein Datentransportproblem, bei dem jedes PE genau k Datenpakete sendet und empfängt.

Satz 2.4. *Ist ein Datentransportproblem durch eine fest vorgegebene k-Permutation definiert, so kann es in $\Theta(nk/b+d)$ Schritten gelöst werden.*

Eine interessante Beobachtung ist, daß für alle oben betrachteten Netzwerke außer Bussen $n/b \in \mathrm{O}(d)$ gilt, so daß sich die Zeit für $k \in \Theta(1)$ auf $\Theta(d)$ vereinfacht.

Schwieriger ist es, eine nicht vorher bekannte Permutation zu bearbeiten (Online routing). Immerhin gibt es effiziente randomisierte Algorithmen:[8)]

Satz 2.5. *Die folgenden Datentransportprobleme lassen sich auf allen hier betrachteten Netzwerken in Zeit $T_{\mathrm{rout}}(k) \in \mathrm{O}(nk/b+d)+\tilde{\mathrm{O}}(\log n/\log\log n+k)$ lösen: (Für $k=1$ wird abkürzend T_{rout} geschrieben.)*

[8)] Für einige Netze gibt es auch effiziente deterministische Algorithmen.

- *Verschicken von k Paketen von jedem PE zu zufällig gewählten Zielen.*
- *Empfangen von k Paketen durch jedes PE von zufällig gewählten Sendern.*
- *Ausführen einer beliebigen k-Permutation. (Bei vollständiger Verknüpfung kann der* $\log n / \log\log n$ *Term wegfallen.)*

Dieses Ergebnis läßt sich ohne asymptotischen Effizienzverlust an andere Situationen anpassen. Wenn nicht alle PEs genau k Pakete senden oder empfangen, können im Zweifel leere Pakete verschickt werden. Sind Nachrichten nichtkonstanter Länge zu verarbeiten, so können diese in viele kleine Pakete aufgeteilt werden. Wenn die zu transportierenden Nachrichten in einem kontinuierlichen Prozeß anfallen, lassen sie sich durch Einfügen geeigneter Synchronisationsoperationen Kommunikationsphasen zuordnen, an deren Ende eine k-Permutation durchgeführt wird. Allerdings sind all diese Maßnahmen in der Praxis eher kontraproduktiv. Von einem robusten Datentransportalgorithmus würde man erwarten, daß sich die beste Effizienz dann ergibt, wenn Pakete dann abgeschickt werden, wenn sie anfallen. In der Praxis ist das auch oft so, läßt sich analytisch aber meist nur schwer nachweisen. Gerade bei den heute weitverbreiteten „Wormhole routing"-Verfahren kann eine Überlastung des Netzwerks durch viele übereilt injizierte lange Nachrichten zu einem deutlichen Rückgang des Durchsatzes führen (UPFAL, 1994). Zwar gibt es theoretische Modelle für die kontinuierliche Erzeugung von Datenpaketen, aber diese sind nur grobe Abstraktionen der realen Situation. Die im folgenden entwickelten Algorithmen machen die optimistische Annahme, daß beim Datentransport keine unerwarteten Verzögerungen auftreten. Leistet eine konkrete Implementierung nicht das Gewünschte, so müssen nachträglich geeignete Maßnahmen getroffen werden (z.B. Flußkontrolle).

2.2.3 Kollektive Operationen

Komplexere Interaktionsmuster zwischen PEs sind die folgenden, immer wieder auftretenden *kollektiven Operationen*:

Satz 2.6. *Sei* $\otimes$ *eine auf Daten der Länge k operierende assoziative und kommutative*[9)] *Operation. Sei* x_i *ein auf PE i liegendes Datum. Es gibt eine Funktion*

$$T_{\text{coll}} \in \begin{cases} \mathrm{O}\big((\log n)^2 + k\big) & \textit{für mehrstufige Netzwerke} \\ \mathrm{O}(nk) & \textit{für Busse} \\ \mathrm{O}(d + k + \log n) & \textit{sonst} \end{cases}$$

so daß die folgenden Operationen in Zeit T_{coll} *durchführbar sind:*

Broadcast: *Versenden einer Nachricht der Länge k an alle PEs.*[10)]

Barrier: *Globale Synchronisation aller PEs.*

Reduktion: *Berechnen von* $\bigotimes_{i<n} x_i$.

Präfixsumme: *Berechnen von* $\bigotimes_{j\le i} x_j$ *auf PE i.*

[9)] Die Kommutativität ist nicht entscheidend, erleichtert aber die Implementierung.

[10)] Auf Bussen ist dies sogar in Zeit $\mathrm{O}(k)$ möglich.

Mehrstufige Netze schneiden relativ schlecht ab, weil die Schaltelemente Nachrichten meist nur weiterleiten, nicht aber verarbeiten können. Deshalb werden diese manchmal durch ein (Baum)-Netzwerk ergänzt, das einige kollektive Operationen in Zeit $O(\log n)$ erledigen kann.

Kollektive Operationen lassen sich durch Austausch von $O(n)$ Nachrichten implementieren, die entlang der Kanten eines in das Netzwerk eingebetteten *Reduktionsbaums* ausgetauscht werden.[11] Eine kollektive Operation beansprucht deshalb für kleine k nur einen asymptotisch verschwindenden Teil der CPU-Leistung. Außer bei Bussen wird auch nur ein asymptotisch verschwindender Teil der Netzwerkbandbreite benötigt. Das heißt, die meisten PEs und die meisten Netzwerkverbindungen können gleichzeitig für andere Dinge verwendet werden. Dies läßt sich allerdings nur nutzen, wenn die kollektiven Operationen auch in einer *asynchronen* Variante zur Verfügung stehen, bei der nach Ablieferung der Operanden nicht auf das Gesamtergebnis gewartet werden muß.

2.2.4 Skalierbarkeit und andere Maßzahlen

Das Prinzip der Leistungsbewertung paralleler Algorithmen ist eigentlich ganz einfach. Es geht um den Vergleich zwischen der *parallelen Ausführungszeit* T_{par} und der *sequentiellen Ausführungszeit* T_{seq}.

Die Tücke liegt allerdings im Detail. So sollte T_{seq} im allgemeinen die Ausführungszeit des besten bekannten sequentiellen Algorithmus zugrundeliegen (AKL, 1989). Was aber, wenn keine Einigkeit herrscht, welcher Algorithmus dies ist oder bessere Algorithmen bekannt sind, diese aber aus praktischen Gründen noch nie implementiert wurden oder zumindest nicht zur Verfügung stehen. Es gibt sogar Probleme, bei denen es für jeden Lösungsalgorithmus immer noch einen besseren Algorithmus gibt (UMLAND UND VOLLMAR, 1992; HOPCROFT UND ULLMAN, 1979). Deshalb wird hier von der praktikableren Konvention ausgegangen, einen anerkannt sinnvollen sequentiellen Algorithmus zu nehmen, der mindestens so sorgfältig programmiert ist wie der parallele Algorithmus. Bei den Messungen in Anhang A genügt es, weitgehend identische Programme für den sequentiellen und parallelen Fall zu verwenden. Allerdings werden die Teile des parallelen Algorithmus, die im sequentiellen Fall sinnlos sind, soweit „abgeklemmt“ sind, daß sie keinen meßbaren Zusatzaufwand verursachen.

T_{seq} ist meist eine durch den Algorithmus bestimmte Funktion von einfachen Probleminstanzparametern. Oft tritt nur ein einziger auf, der dann *Problemgröße* genannt wird. T_{par} kann zusätzlich von den Parametern der Maschine (n, d, b, …) und zusätzlichen Probleminstanzparametern, wie dem *sequentiellen Anteil* oder der *Korngröße (Granularity)*, abhängen.[12]

Um die in T_{par} und T_{seq} enthaltene Information übersichtlich aufbereiten zu können, sind eine Vielzahl von Maßzahlen entwickelt worden, von denen hier nur einige verwendet werden. Die *Beschleunigung (Speedup)* ist ein Maß für die durch die Parallelisierung erreichte Leistungssteigerung.

Definition 2.7 (Beschleunigung). $S := \frac{T_{\text{seq}}}{T_{\text{par}}}$.

Eine hohe Beschleunigung wird nur dann erreicht, wenn die *PE-Auslastung*, d.h., die Zahl beschäftigter PEs, hoch ist, und die durchgeführte Arbeit auch „nützlich“ ist – also zur Lösung des Problems beiträgt.

Wird die Beschleunigung in Relation zum Aufwand an PEs gesetzt, so ergibt sich ein Maß für die Kosteneffizienz des Verfahrens.

[11] Daß auch für Präfixsummen $O(n)$ Nachrichten ausreichen, ist anscheinend nicht allgemein bekannt. Aufgabe 1.38 in LEIGHTON (1992) enthält allerdings einen deutlichen Hinweis.

[12] Wie sequentieller Anteil und Korngröße genau zu definieren sind, hängt natürlich vom Anwendungsmodell und dem verwendeten Parallelisierungsansatz ab.

Definition 2.8 (Effizienz). $E := \frac{T_{\text{seq}}}{nT_{\text{par}}}$

Erstrebenswert ist eine Effizienz nahe bei 1. Selbst der Fall $E > 1$ (die sogenannte superlineare Beschleunigung) kommt vor. Manchmal ergibt sich daraus ein verbesserter sequentieller Algorithmus, der von der Emulation des parallelen Algorithmus auf einem sequentiellen Rechner ausgeht.[13)] In Abschnitt A.5.2 wird ein solcher Fall für das Rucksackproblem näher diskutiert.

Es ist erstaunlich schwer, genau zu charakterisieren, was ein effizienter paralleler Algorithmus ist. Nach *Amdahl's Gesetz* ist bei festgehaltener Problemgröße nämlich jeder parallele Algorithmus asymptotisch ineffizient, da der sequentielle Anteil des Problems die erreichbare Beschleunigung begrenzt (HWANG, 1993). Ist bei geeigneter Skalierung der Problemgröße mit der Prozessorzahl dagegen eine unbeschränkte Beschleunigung erreichbar, so kann im weiteren Sinne von einem *skalierbaren Algorithmus* gesprochen werden. Noch besser ist es, wenn zusätzlich eine konstante, d.h. nicht mehr von n abhängige, Effizienz erreichbar ist. Selbst wenn das möglich ist, kann aber noch nicht unbedingt von „guter" Skalierbarkeit gesprochen werden. Es kommt zusätzlich darauf an, wie schnell die Problemgröße mit n erhöht werden muß, um konstante Effizienz zu erreichen.

Hier wird als formales Endergebnis die folgende aufbereitete Darstellung der parallelen Ausführungszeit angestrebt:

$$T_{\text{par}} \leq (1+\varepsilon)\frac{T_{\text{seq}}}{n} + T_{\text{rest}}$$

Dabei ist T_{rest} bei geeigneter Skalierung der Problemparameter eine im Vergleich zu T_{seq}/n langsam wachsende Funktion, die meist nur asymptotisch angegeben wird. $\varepsilon \in \mathbb{R}_+$ ist eine beliebig kleine Konstante.

Bei randomisierten Algorithmen kann der zu T_{seq} proportionale Mehraufwand, der sich hinter dem ε verbirgt, auch eine Zufallsvariable sein. Dann kann statt $1+\varepsilon$ ein Term wie $1+\varepsilon\tilde{O}(1)$ auftreten. An dieser Stelle wird ε dann wie eine Variable behandelt, damit es nicht als konstanter Faktor unter den Tisch fällt. Jedenfall ist für hinreichend große Probleminstanzen mit hoher Wahrscheinlichkeit eine Effizienz beliebig nahe eins erreichbar, und T_{rest} ist ein Maß für die Skalierbarkeit des Algorithmus. Im allgemeinen hängt T_{rest} von ε ab. Da ε konstant ist und T_{rest} meist nur bis auf konstante Faktoren hergeleitet wird, ist diese Abhängigkeit in den Ergebnissen allerdings nicht explizit sichtbar. Wenn nicht anders erwähnt, lassen sich die Ergebnisse so modifizieren, daß ε als eine von n unabhängige zusätzliche Variable aufgefaßt wird. T_{rest} erhält dann meist einen zusätzlichen Faktor von $\Theta\left(\frac{1}{\varepsilon}\right)$.

2.3 Anwendungsmodell

Um eine Diskussionsgrundlage zu haben, wird in Abschnitt 2.3.1 zunächst das Anwendungsmodell *Baumförmige Berechnung* in möglichst konziser Weise dargestellt. Erst danach wird in Abschnitt 2.3.2 genauer erklärt, was das mit Parallelisierung von Tiefensuche zu tun hat. Abschnitt 2.3.3 diskutiert dann einen wichtigen Aspekt paralleler Suche, der bewußt aus dem Modell herausgehalten wurde. Die abstrakte Diskussion wird konkretisiert, indem in Abschnitt 2.3.4 tatsächliche Anwendungen beschrieben werden, für die baumförmige Berechnungen ein geeignetes Modell darstellen. Eine noch detailliertere Diskussion einiger konkreter Beispiele findet sich in Abschnitt A.2. Schließlich wird in Abschnitt 2.3.5 ein Bezug zu anderen existierenden Anwendungsmodellen hergestellt.

[13)] Leider geht das nicht immer, da auch Effekte wie die größere Speicherkapazität des Parallelrechners Ursache für die hohe Leistung sein können.

2.3.1 Baumförmige Berechnungen

Definition 2.9 (Wurzelproblem). Ausgangspunkt einer *baumförmigen Berechnung* ist eine Beschreibung des Wurzelproblem, P_{root}, das zu Beginn auf PE 0 liegt.

Aus dem Wurzelproblem können durch Aufspaltung in zwei[14] Teile neue *Teilprobleme* geschaffen werden. Außerdem kann ein existierendes Teilproblem durch sequentielle Bearbeitung über einen Zeitraum t in ein anderes (nämlich um t „kleineres“) Teilproblem umgewandelt werden.

Definition 2.10 (Teilproblem). $\mathcal{P}$ bezeichnet die Menge der von P_{root} ableitbaren *Teilprobleme*. P_{root} ist ein Teilproblem. Die Operation $\text{work}(P,t) \in \mathcal{P}^{\mathcal{P}\times(\mathbb{R}_*\cup\{\infty\})}$ erzeugt aus einem Teilproblem P ein neues Teilproblem P'. Die *Spaltoperation* $\text{split}(P)$ erzeugt aus einem Teilproblem P zwei neue Teilprobleme P_1 und P_2. Nichts anderes sind Teilprobleme.

Das Wort „Operation“ wird hier bewußt an Stelle von „Funktion“ verwendet, um deutlich zu machen, daß das Teilproblem „verbraucht“ und durch ein neues ersetzt wird.

Die Schwierigkeit oder *Größe* eines Teilproblems wird durch seine sequentielle Ausführungszeit festgelegt, da dies ein für alle Anwendungen anwendbares Maß ist.

Definition 2.11 (Problemgröße). Die Funktion $T \in {\mathbb{R}_*}^{\mathcal{P}}$ gibt die Zeit an, die notwendig ist, ein Teilproblem durch sequentielles Rechnen abzuarbeiten. $T(P)$ wird als *Problemgröße* von P bezeichnet.

Definition 2.12 (Leeres Teilproblem). Ein Teilproblem P mit $T(P) = 0$ heißt *leeres Teilproblem*. $P_\emptyset$ wird als Symbol für ein leeres Teilproblem verwendet.

Zu beachten ist, daß die Funktion T der Implementierung nicht zur Verfügung steht. Nur leere Probleme sind als solche erkennbar. Nun sind die Begriffe eingeführt, die nötig sind, um die Eigenschaften der Operation „work“ zu präzisieren.

Definition 2.13 (Sequentielles Bearbeiten). Die Anwendung der Operation $\text{work}(P,t)$ kostet Zeit $\min(t, T(P))$, und es gilt:

$$T(\text{work}(P,t)) = \max(0, T(P) - t) \quad .$$

Die Operation „work“ stellt also eine Schnittstelle zu einem sequentiellen Algorithmus dar, mit dessen Hilfe die Problemgröße eines beliebigen Teilproblems reduziert werden kann. Insbesondere ist die sequentielle Ausführungszeit mit der Problemgröße des Wurzelproblems gleichzusetzen.

$$T_{\text{seq}} = T(P_{\text{root}})$$

Entscheidend für die Parallelisierung ist nun das Verhalten der Spaltoperation.

Definition 2.14 (Spalten). Die Durchführung der Spaltoperation kostet Zeit T_{split}, und es gilt

$$\text{split}(P) = (P_1, P_2) \implies T(P) = T(P_1) + T(P_2) \quad .$$

[14] Der allgemeinere Fall einer Aufteilung in k Teile ist für die Theorie nicht sehr interessant, wird allerdings in Abschnitt A.4.2 kurz aufgegriffen, da einige Algorithmen dann etwas effizienter und flexibler implementiert werden können.

Das Ausgangsproblem wird also unter den Spaltergebnissen aufgeteilt, ohne daß sequentieller Aufwand wegfällt[15] oder hinzukommt[16]. Sonst ist nichts über die Ergebnisse einer einzelnen Spaltoperation bekannt. Selbst ein leeres Problem kann entstehen. Um trotzdem zu einer sinnvollen Analyse zu gelangen, müssen allerdings bestimmte Mindestanforderungen an die Spaltoperation gestellt werden. Würde z.B. immer ein leeres Problem abgespalten, wäre ja keine Parallelisierung möglich. Es wird deshalb gefordert, daß nach einer Maximalzahl von h Spaltungen eine atomare Problemgröße T_{atomic} unterschritten wird.

Definition 2.15 (Generation). Die *Generation* $\text{gen}(P)$ eines Teilproblems gibt an, durch wieviele Spaltoperationen P entstanden ist. Also $\text{gen}(P_{\text{root}}) = 0$ und

$$\text{split}(P) = (P_1, P_2) \implies \text{gen}(P_1) = \text{gen}(P_2) = 1 + \text{gen}(P) \quad .$$

Definition 2.16 (Maximale Spalttiefe).

$$h := \min\left\{h' \in \mathbb{N} \,|\, \forall P \in \mathcal{P} : \text{gen}(P) \geq h' \implies T(P) \leq T_{\text{atomic}}\right\} \quad .$$

Die maximale Spalttiefe h ist ein Maß für die Irregularität der Berechnung. Je größer h, desto mehr Spaltoperationen können nötig sein, um die Last vernünftig ausgleichen zu können. T_{atomic} ist dagegen ein Maß für die Problemgranularität. Hat ein Problem diese Größe unterschritten, kann es u.U. nicht mehr aufgespalten werden und muß sequentiell abgearbeitet werden. Damit überhaupt eine effiziente Parallelisierung denkbar ist, muß offenbar $T_{\text{seq}} \geq nT_{\text{atomic}}$ gelten. Diese Annahme wird im folgenden zwecks Vereinfachung der Diskussion durchgehend gemacht.

Schließlich sei ein Maß für den Platzbedarf eines Teilproblems eingeführt. Dies dient hier vor allem dazu, den Aufwand für das Versenden von Teilproblemen zu anderen PEs zu erfassen.

Definition 2.17 (Nachrichtenlänge). l sei eine obere Schranke für die Länge der Repräsentation eines beliebigen Teilproblems.

Die parallele Ausführung einer baumförmigen Berechnung ist im Prinzip sehr einfach. Durch Aufspalten und Versenden von Teilproblemen kann ein hoher Grad an Parallelität erreicht werden. Kosten für das Einsammeln der Ergebnisse fallen im abstrakten Modell nicht an (siehe auch die Diskussion in Abschnitt 2.3.3). Die Schwierigkeit besteht vor allem darin, möglichst gute Prozessorauslastung bei vertretbarem Kommunikationsaufwand zu erzielen, ohne verläßliche Information über die Größe von Teilproblemen zu besitzen.

Die doch recht große Zahl an Parametern läßt sich bei Bedarf reduzieren. In Abschnitt A.5.1 zeigt sich z.B., daß h, l und T_{split} oft das gleiche asymptotische Verhalten zeigen. In der Literatur wird oft die Annahme $\left\{T_{\text{split}}, T_{\text{atomic}}, l\right\} \subseteq \Theta(1)$ gemacht. Das ist zwar bestenfalls eine Annäherung an reale Gegebenheiten, läßt aber immer noch interessante Aussagen zu.[17] Zusätzlich wird oft $h \in \Theta\left(\log T_{\text{seq}}\right)$ angenommen. Ein noch langsamer wachsendes h ist übrigens nicht möglich.

[15] Oft wird als Nebeneffekt der Aufspaltung ein wenig sequentielle Arbeit geleistet. Diese ist aber im allgemeinen vernachlässigbar – vor allem wenn es um obere Schranken geht.

[16] Inwieweit diese Annahme gerechtfertigt ist, wird in Abschnitt 2.3.3 ausführlich diskutiert.

[17] Aus ähnlichen Gründen wird oft auch das Maschinenmodell vereinfacht, indem eine vollständige Verknüpfung angenommen wird.

Die Definitionen 2.14 und 2.16 sind nur konsistent, wenn $h \geq \log \frac{T_{\text{seq}}}{T_{\text{atomic}}}$. Zusammen mit der Annahme $T_{\text{seq}} \geq nT_{\text{atomic}}$ ergibt sich daraus

$$h \geq \log n \tag{2.1}$$

2.3.2 Bezug zu Tiefensuche

Hier wird erklärt, wie baumförmige Berechnungen Tiefensuchalgorithmen modellieren können. In Abschnitt A.4 wird diese allgemein gehaltene Darstellung durch einige konkrete Beispiele ergänzt. Abbildung 2.2 zeigt einen generischen Algorithmus für nichtrekursive sequentielle Tiefensuche. Zu Beginn wird der Stapel mit der Suchbaumwurzel initialisiert. In einer Schleife werden dann Suchbaumknoten von der Spitze des Stapels entnommen und ausgewertet. Blätter können zur Lösung beitragen oder einfach Sackgassen sein. Innere Knoten werden expandiert und die so bestimmten Nachfolger auf dem Stapel abgelegt. Die Schleife wird solange durchlaufen bis der Stapel leer ist.

```
let stack consist of root node only
while stack is not empty do
    remove a node N from the stack
    if N is a leaf then
        evaluate leaf N
    else
        put successors of N on the stack
    fi
```

Abbildung 2.2: Abstrakter Tiefensuchalgorithmus.

Viele Algorithmen beruhen im Kern auf diesem Prinzip, auch wenn dies nicht immer auf den ersten Blick erkennbar ist. Die Kontrollstruktur ist meist eng mit der Anwendung verflochten, und auf dem Stapel werden nicht immer Knoten abgelegt, sondern nur Unterschiede zwischen Knoten oder gar nur eine Identifikation des ersten noch nicht betrachteten Nachfolgers. (Bei rekursiven Implementierungen ist diese Identifikation z.B. die Aufrufumgebung eines Programmteils, das den nächsten Nachfolger erzeugt.) Eine Schlüsselaufgabe des Stapels ist jedenfalls die Repräsentation des Pfades von der Wurzel bis zum gerade betrachteten Knoten und der dort abzweigenden noch nicht besuchten Teilbäume. Der obere Teil von Abbildung 2.3 zeigt dies in einer schematischen Darstellung.

Eine Übersetzung von Tiefensuche in baumförmige Berechnungen ist nun relativ einfach.

Teilproblem: Der gesamte Zustand einer sequentiellen Suche (Suchkontext) einschließlich des Stapels, der den Pfad von der Wurzel bis zum gerade bearbeiteten Knoten repräsentiert und außerdem Informationen darüber enthält, welche Zweige des Suchbaums noch abzuarbeiten sind.

Wurzelproblem: Ein Suchkontext mit bis auf den Wurzelknoten leerem Stapel.

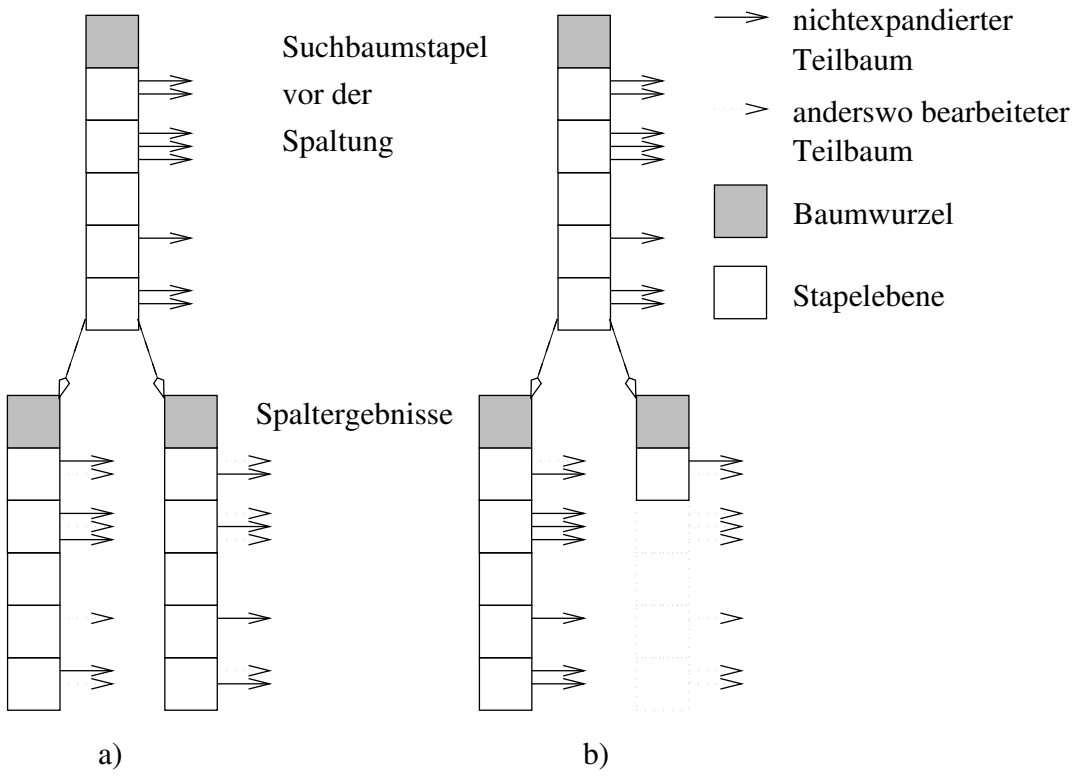

Abbildung 2.3: Zwei Stapelaufteilungsstrategien. a): Spalten auf allen Ebenen. b): Möglichst weit oben im Stapel.

Operation „work“: Sequentielle Tiefensuche in einem Suchkontext für einen vorgegebenen maximalen Zeitraum.

Operation „split“: Aufspalten des Suchkontexts in zwei Suchkontexte, die disjunkte Teile des Suchbaums repräsentieren. Schlüssel dazu ist die Aufteilung der offenen Alternativen auf dem Stapel. Abbildung 2.3 skizziert zwei wichtige Möglichkeiten. Oben ist der gegenwärtige Zustand des Stapels mit einem Pfeil für jeden undurchsuchten Zweig dargestellt. Unten steht das Spaltergebnis. Gepunktet eingezeichnete Pfeile stellen Suchbaumzweige dar, die anderswo bearbeitet werden. Am häufigsten verwendet wird die bei b) abgebildete Abgabe eines (oder mehrerer) Zweige ganz oben im Stapel. Dort erwartet man am ehesten große undurchsuchte Teilbäume. Nach der Spaltoperation ist der „linke“ Stapel eine Kopie des ursprünglichen Stapels, außer daß die abgegebenen Zweige als solche markiert sind und hier nicht mehr bearbeitet werden müssen. Der „rechte“ Stapel ist oberhalb der Aufspaltungsebene ebenfalls eine Kopie. In der Aufspaltungsebene sind die Zweige komplementär zum linken Stapel markiert, so daß nur die tatsächlich abgegebenen Zweige von einer folgenden sequentiellen Bearbeitung weiterverfolgt werden. Etwas robuster ist die bei a) abgebildete Methode, jeden zweiten Zweig auf jeder Ebene des Stapels abzugeben (RAO UND KUMAR, 1987a).

Problemgröße T: Sequentieller Zeitaufwand für das Durchsuchen eines durch einen Suchkontext repräsentierten Teilbaums.

Man beachte, daß tatsächlich i. allg. keine Möglichkeit besteht, $T(P)$ zur Laufzeit zu berechnen

oder auch nur verläßlich abzuschätzen. Oft wäre eine genaue Bestimmung von $T(P)$ gleichbedeutend mit einem kompletten Durchlaufen des Teilbaums.

Maximale Spalttiefe h: Ist der Verzweigungsgrad des Suchbaums durch eine Konstante beschränkt und wird die einfache Strategie des Abspaltens von Teilbäumen nahe der Wurzel angewandt, so ist h proportional zur maximalen Tiefe des Suchbaums – eine Größe, die sich oft leicht abschätzen läßt.

Nachrichtenlänge l: Maximaler Platzbedarf einer (u.U. komprimierten) Beschreibung eines Suchkontextes.

Granularität T_{atomic}: Meist eine obere Schranke für den Aufwand von Blattbewertung und Knotenexpansion.

2.3.3 Abhängigkeiten zwischen Teilproblemen

Die wichtigste Einschränkung baumförmiger Berechnungen findet sich in Definition 2.14. Es wird davon ausgegangen, daß Teilprobleme völlig unabhängig voneinander auf beliebigen PEs ausgewertet werden können, ohne daß die insgesamt zu leistende sequentielle Arbeit dadurch beeinflußt wird. Wörtlich stimmt dies eigentlich nie. Zumindest müssen Ergebnisse der Teilberechnungen zu einem Gesamtergebnis zusammengesetzt werden. Da dieses Zusammensetzen aber in analoger Weise wie das Verteilen der Teilprobleme stattfinden kann, ist nicht zu erwarten, daß dadurch irgendwelche zusätzlichen Komplikationen entstehen. Oft ist die Zusammensetzoperation sogar assoziativ (z.B. max, min, $+$, $\cup$, $\cap$, $\wedge$), und die Ergebnisbestimmung kann dann zum Ende der Berechnung durch eine einzige globale Reduktionsoperation ausgeführt werden.

Schwerwiegender ist die Annahme aus Definition 2.14, daß die Summe der Teilproblemgrößen gleich der ursprünglichen Problemgröße ist. Durch Auswerten des einen Teilproblems können nämlich Informationen bekannt werden, die das andere Teilproblem beeinflussen (meistens verkleinern). Trotzdem gibt es wichtige Klassen von Anwendungen, bei denen dieser Effekt nicht auftritt:

Vollständiges Durchlaufen des Suchbaums: Soll eine Funktion berechnet werden, die in jedem Fall von allen Blättern des Suchbaums abhängt, so muß dieser komplett durchlaufen werden. Typische Beispiele sind: Bestimme alle Lösungen eines Problems, zähle alle Lösungen, berechne Durchschnittswerte usw. Das bedeutet nicht, daß Suchraum und durchsuchter Raum identisch sein müssen. Solange Beschneidungen des Suchraums nicht von der Reihenfolge der Berechnungen abhängen, genügt es, den Suchbaum als den durchsuchten Teil des Suchraums zu definieren.

Widerlegung: Ein besonders häufig auftretender Fall des vollständigen Durchlaufens des Suchbaums ist die Aufgabe nachzuweisen, daß kein Blatt des Suchbaums eine bestimmte Eigenschaft hat. Dies ist zum Beispiel bei einigen Theorembeweisern der Fall. Interessant ist auch die folgende Einsatzvariante von Branch-and-bound.[18] Eine im allgemeinen gut funktionierende Heuristik habe eine Näherungslösung mit Bewertung w für ein Problem geliefert. Nun soll nachgewiesen werden, daß die Lösung um nicht mehr als k Punkte vom (unbekannten) Optimum entfernt liegt. Dazu wird ein Branch-and-bound Algorithmus mit dem Auftrag gestartet, eine Lösung zu finden, die eine bessere Bewertung als $w-k$ hat ($w+k$ falls maximiert wird). Wird, wie erwartet, keine solche Lösung gefunden, ist der Nachweis erbracht. Eine ähnliche Situation tritt auch bei allgemeinen Branch-and-bound-Problemen häufig auf.

[18] Branch-and-bound basiert auf einer heuristischen Funktion, die jedem Knoten (des Suchraums) eine optimistische Abschätzung der besten daraus ableitbaren Lösung zuordnet. Knoten, deren Bewertung schlechter als die der besten bereits bekannten Lösung ist, brauchen deshalb nicht weiter betrachtet werden. (Diese Definition würden einige Autoren als zu eng bezeichnen; sie entspricht aber durchaus dem Standard CLAUS UND SCHWILL (1988)).

Oft wird die optimale Lösung relativ schnell gefunden, und der Hauptteil der Suche dient dann dem Nachweis der Optimalität. Diese Endphase ist dann wieder eine Widerlegungssuche.

Iteratives Vertiefen: Ist ein Suchraum zu groß, um vollständig durchsucht zu werden, so hilft oft *iteratives Vertiefen*: Die Suchtiefe (oder eine mit der Suchtiefe streng monoton wachsende Funktion) wird auf einen Maximalwert begrenzt. Ist die Suche deshalb erfolglos, wird dieser Maximalwert erhöht und die Suche erneut gestartet. Diese Vertiefung wird solange wiederholt, bis eine Lösung gefunden wird. Bei Theorembeweisern dient dies der Begrenzung des unendlich großen Suchraums (z.B. ERTEL (1992)). Beim *IDA*-Algorithmus* (KORF, 1985) wird dadurch (zusätzlich) die Optimalität der Lösung erzwungen. Alle bis auf die letzte Iteration sind deshalb Widerlegungssuchen, die meist keine Baumbeschneidungen aufweisen.

Häufig besteht die einzige Abhängigkeit zwischen Teilproblemen darin, daß die Suche gestoppt wird, sobald die erste Lösung gefunden wurde. (Zum Beispiel gilt dies für die letzte Iteration von vielen Algorithmen, die auf iterativem Vertiefen beruhen). Oft ist keine gute Heuristik bekannt, die voraussagt, welches Teilproblem wahrscheinlich schneller zu einer Lösung führt. Für diesen Fall weisen Modelle wie in RAO UND KUMAR (1993) darauf hin, daß bei gleichmäßiger Verteilung der Lösungen der von paralleler Tiefensuche durchsuchte Baum im Mittel genauso groß ist wie im sequentiellen Fall. Bei ungleichmäßiger Verteilung ist sogar eine mittlere Effizienz größer eins möglich. Für ein verwandtes Modell konstruiert ERTEL (1992) sogar ein Klasse von Problemen, die eine Beschleunigung von $S \in \Theta(2^n)$ aufweisen! Die Strategie, sich nur um eine gute Lastverteilung zu kümmern, kann also in Abwesenheit besserer Heuristiken ein erfolgreicher Ansatz sein. In diesem Fall sind baumförmige Berechnungen ein gutes Modell – auch wenn sie die Anwendung nicht vollständig modellieren.

Schließlich gibt es Suchprobleme, bei denen sich eine Abhängigkeit zwischen Teilproblemen nicht wegdiskutieren läßt. Zum Beispiel führt die $\alpha\beta$-Heuristik bei Spielbaumsuchalgorithmen dazu, daß Baumbeschneidungen vom Suchergebnis für Geschwisterknoten abhängen. Wird diese Tatsache vernachlässigt, ergeben sich um ein Vielfaches vergrößerte Suchbäume. Trotzdem beruhen die besten bekannten Lastverteilungsverfahren für Spielbaumsuche im Kern auf Algorithmen, die für baumförmige Berechnungen besonders effizient sind (FELDMANN, 1993; FELDMANN ET AL., 1994; HOPP UND SANDERS, 1995). Selbst für Anwendungen mit starken Abhängigkeiten zwischen Teilproblemen können baumförmige Berechnungen also ein gutes Modell für den Lastverteilungsaspekt der Parallelisierung sein.

2.3.4 Bezug zu konkreten Problemen

Wegen ihrer Einfachheit werden klassische Probleme aus der Unterhaltungsmathematik wie das *n*-Damen-Problem, das Springertour-Problem (FINKEL UND MANBER, 1987), das *Blocks in a box puzzle* (FRYE UND MYCZKOWSKI, 1991) und und vor allem das 15-Puzzle (bzw. *n*-Puzzle) als Benchmark für Suchalgorithmen verwendet (KORF, 1985; KUMAR ET AL., 1994; REINEFELD UND SCHNECKE, 1994; GASSER, 1995).

Auch „ernsthafte“ mathematische Probleme lassen sich in einigen Fällen durch heuristisches Backtracking lösen. In diese Kategorie fallen die Suche nach zustandsminimalen Übergangstabellen für Zellularautomaten (SANDERS, 1993, 1994d) und die Suche nach Golomblinealen (Abschnitt A.2, DOLLAS ET AL. (1995)).

Ein weites Gebiet sind Algorithmen zur exakten Lösung **NP**-vollständiger Probleme aller Art. Im Prinzip läßt sich für jedes solches Problem sofort ein Tiefensuchalgorithmus angeben, indem ein nichtdeterministischer Algorithmus auf einer deterministischen Maschine emuliert wird.[19)] Obwohl praktisch einsetzbare Algorithmen meist eine Vielzahl zusätzlicher Heuristiken enthalten, bleiben sie oft doch als Tiefensuchalgorithmen charakterisierbar. Beispiele sind die Lösung schwerer Instanzen des

[19)] Aus nichtdeterministischen Entscheidungen werden dann Verzweigungen des Suchbaums.

Erfüllbarkeitsproblems SAT (BÖHM UND SPECKENMEYER, 1994), das 0/1-Rucksackproblem (Abschnitt A.2), Constraint satisfaction (LIN UND YANG, 1995) oder Testmustergenerierung für integrierte Schaltkreise (ARVINDAM ET AL., 1990a).

Weitere „klassische" Aufgabenbereiche, für die parallele Tiefensuche einsetzbar ist, sind Theorembeweiser (ERTEL, 1992), Expertensysteme sowie parallele funktionale (AHARONI ET AL., 1993) und logische Programmiersprachen (LIN, 1991; KERGOMMEAUX UND CODOGNET, 1994).

Schließlich gibt es auch Anwendungen, die wenig mit Tiefensuche zu tun haben und trotzdem durch baumförmige Berechnungen modelliert werden können. Dies gilt für Teile-und-herrsche-Algorithmen, bei denen die Größe der produzierten Teile nicht bekannt ist und bei denen mit der Problemaufteilung keine Teilung größerer Datenmengen einhergeht. (In diesem Fall wäre die Annahme eines teilproblemunabhängigen l nicht mehr gerechtfertigt.) Ein Beispiel ist der in CHAKRABARTI ET AL. (1994) beschriebene Algorithmus zur Eigenwertbestimmung tridiagonaler Matrizen, adaptive numerische Integrationsverfahren (DE DONCKER ET AL., 1996) oder das Bestimmen der Extrema einer nichtlinearen Funktion in NONNENMACHER UND MLYNSKI (1996). Sehr interessant ist der Fall von Simulationen realer Systeme, die von Natur aus baumförmig sind. So wird in CASARI ET AL. (1994) ein Algorithmus zur Simulation des Verhaltens hochenergetischer Teilchen in Materie parallelisiert. Der bestimmende Effekt ist dabei der Zerfall eines Teilchens in Sekundärteilchen, die ihrerseits weiter zerfallen können, usw. Es kann auch nützlich sein, Anwendungen, die scheinbar leicht zu parallelisieren sind, wie z.B. Ray tracing, das Schema baumförmiger Berechnungen „überzustülpen". Es wird dann zwar nicht jede Optimierungsmöglichkeit genützt, aber der Implementierungsaufwand wird verringert und das entstehende Programm ist gut gerüstet, um mit Lastverteilungsproblemen fertig zu werden, die durch die Arbeitsumgebung verursacht sind. (Unterschiedliche Leistungsfähigkeit der Maschinen und ihrer Verbindungen; durch andere Jobs verursachte Schwankungen der Verfügbarkeit von Kommunikationsbandbreite und CPUs.) Einige der in BLUMOFE ET AL. (1995) untersuchten Anwendungen sind von diesem Typ.

2.3.5 Bezug zu anderen abstrakten Modellen

Baumförmige Berechnungen lassen sich als Synthese aus den folgenden in der Literatur verwendeten Modellen auffassen: (Abgesehen von der weniger wichtigen Tatsache, daß die Parameter l, T_{split} und T_{atomic} meist als konstant angesehen werden.)

α-Spalten: Dieses in RAO UND KUMAR (1987b) eingeführte Modell führt den Begriff des Teilproblems und der Spaltoperation ein. Allerdings muß die Spaltoperation stärkeren Anforderungen genügen. Ist $(P_1, P_2) = \text{split}(P)$, so muß garantiert sein, daß $\min(T(P_1), T(P_2)) \geq \alpha T(P)$ für eine feste Konstante $\alpha \in (0, 1/2]$. Dies bedeutet

$$h \leq \log_{\frac{1}{1-\alpha}} \frac{T_{\text{seq}}}{T_{\text{atomic}}} \quad .$$

Die Spalttiefe h ist also immer logarithmisch in $T_{\text{seq}}/T_{\text{atomic}}$. Leider ist keine Spaltoperation bekannt, die dies für beliebige zugrundeliegende Suchbäume garantiert. (Auch für die recht robuste Funktion in Abbildung 2.3-a) gibt es Gegenbeispiele.) Außerdem verlangt α-Spalten, daß die Spaltoperation eine gleichmäßige Minimalqualität garantiert. „Ausrutscher" oder gar systematisch schlechtes Verhalten über eine Anzahl aufeinanderfolgender Spaltoperationen sind nicht zugelassen. Trotzdem wird im folgenden auf α-Spalten zurückgegriffen, wenn Algorithmen diskutiert werden, die für gleichmäßig wirksame Spaltoperationen deutlich besser funktionieren als für allgemeine baumförmige Berechnungen.

Implizit definierte Bäume: In etwas theoretischer ausgelegten Arbeiten wie LEIGHTON ET AL. (1989) wird das abstrakte Problem betrachtet, einen implizit (durch eine Knotenex-

pansionsfunktion) definierten (Binär-)Baum zu traversieren oder in ein PE-Netzwerk einzubetten.

Dieses Modell läßt sich auf Tiefensuche anwenden (z.B. RANADE (1994)), indem davon ausgegangen wird, daß Suchbaum und einzubettender Baum identisch sind. In erster Näherung lassen sich baumförmige Berechnungen und die damit verbundene Einführung einer Spaltoperation als eine Abstraktion auffassen, die es erlaubt, einen Suchbaum auf geschickte Weise in einen „Spaltbaum“ zu transformieren, der auf jeden Fall ein Binärbaum ist und u.U. zusätzlich flacher und ausgewogener als der ursprüngliche Suchbaum ist. Außerdem bieten baumförmige Berechnungen die Möglichkeit, beliebig zwischen sequentieller Bearbeitung und Aufspalten eines Teilproblems umzuschalten und erlauben dadurch erst die direkte Formulierung der in Abschnitt 2.4.2 eingeführten *empfängerveranlaßten* Lastverteilungsalgorithmen, die auf realen Maschinen deutlich effizienter sind als die *senderveranlaßten*[20] Algorithmen, die sich bei alleiniger Verfügbarkeit der Operation *expandiere Knoten* anzubieten scheinen.

Schließlich seien noch einige Modelle erwähnt, die andere Ziele verfolgen, aber trotzdem eine gewisse Verwandtschaft mit baumförmigen Berechnungen aufweisen. In BLUMOFE UND LEISERSON (1994) werden sogenannte *fully strict multithreaded computations* untersucht. Baumförmige Berechnungen lassen sich bei Vernachlässigung der Parameter T_{split}, T_{atomic} und l in dieses Modell übersetzen, das außerdem noch die Modellierung bestimmter vorhersagbarer Abhängigkeiten zwischen Teilproblemen erlaubt.

Die in Abschnitt 2.3.3 beschriebenen Abhängigkeiten, die zu Beschneidungen des Suchbaums führen können, sind allerdings nicht vorhersagbar und werden deshalb oft stochastisch modelliert (KARP, 1983; PEARL, 1984; HELMAN, 1988; DION ET AL., 1994). Der Argumentation aus Abschnitt 2.3.3 folgend, daß baumförmige Berechnungen den Lastverteilungsaspekt beim Durchsuchen dieser stochastisch erzeugten Bäume modellieren können, interessiert in diesem Zusammenhang die Struktur solcher Bäume – insbesondere ihre Tiefe. Interessanterweise ergibt sich für viele stochastische Modelle $h \in \tilde{\mathrm{O}}\left(\log T_{\text{seq}} / T_{\text{atomic}}\right)$, also relativ flache Bäume. Sehr unregelmäßige Probleminstanzen lassen sich also so nicht modellieren.

2.4 Ansätze zur Parallelisierung

Aus den vorangegangen Abschnitten ist bereits ablesbar, daß es eine Vielzahl von Arbeiten über parallele Suche gibt. (Übersichtsartikel finden sich zum Beispiel in ARVINDAM ET AL. (1990a), KUMAR UND ANANTH (1991), KHAMBEKAR (1992) und KUMAR ET AL. (1994).) Um den Umfang dieser Arbeit in Grenzen zu halten, sollen hier nur zwei wichtige Klassen von Parallelisierungsansätzen vorgestellt werden, die aber Bestandteil der meisten erfolgreichen Algorithmen sind und auch als Grundlage für die hier betrachteten Algorithmen dienen können.

2.4.1 Mehrfaches Spalten des Wurzelproblems

Die Bereitstellung von Parallelität ist bei baumförmigen Berechnungen im Prinzip kein Problem. Durch Anwendung der Spaltoperation können beliebig viele unabhängige Teilprobleme erzeugt werden, und nur die atomare Granularität T_{atomic} setzt eine Grenze für die Feinkörnigkeit der Berechnung. Viele Parallelisierungsansätze gehen deshalb den naheliegenden Weg, das Wurzelproblem durch wiederholtes Spalten in hinreichend viele Teilprobleme zu zerlegen und diese dann ohne weiteres Spalten abzuarbeiten.

[20] Hier entscheidet der Absender, wann er Probleme aufspaltet und Last abgibt.

Im einfachsten Fall erhält jedes PE genau ein Teilproblem (z.B. EL-DESSOUKI UND HUEN (1980)). Dies läßt sich mit minimaler Kommunikation erreichen. Abbildung 2.4 zeigt eine Variante eines solchen Algorithmus für den Fall, daß n eine Zweierpotenz ist. Das Wurzelproblem wird durch einen Broadcast verteilt und wiederholt gespalten, wobei abhängig von den Bits der PE-Nummer genau eines der erzeugten Teilprobleme behalten wird. Nach $\log n$ dieser Operationen ist die anfängliche Symmetrie vollständig gebrochen. Abbildung 2.5 zeigt dies anhand eines Beispiels für vier PEs. Die Teilprobleme können nun unabhängig voneinander abgearbeitet werden. Verallgemeinerungen für beliebige n und für die direkte Behandlung von Bäumen mit variablem Knotengrad sind möglich (EL-DESSOUKI UND HUEN, 1980; SANDERS, 1993; HENRICH, 1994).

```
(* Let n = 2^k. *)
var P, P_1, P_2 : subproblem
P := P_root                                   (* implies broadcast of root problem *)
for j := 0 to k − 1 do                        (* for each bit of i_PE. *)
    (P_1, P_2) := split(P)
    P := if (i_PE bitand 2^j) = 0 then P_1 else P_2
endfor
P := work(P, ∞)
```

Abbildung 2.4: Einfacher statischer Lastverteilungsalgorithmus.

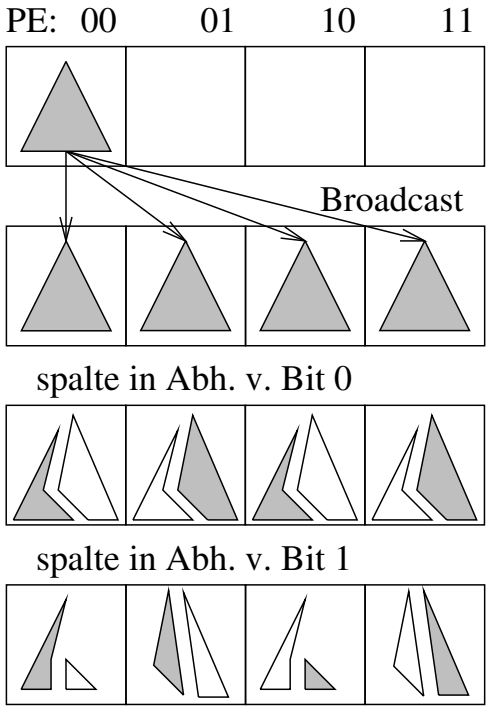

Abbildung 2.5: Einfache statische Lastverteilung für $n = 4$.

Wegen der ungleichmäßigen Aufspaltung führt dies aber zu sehr niedriger Effizienz. (Näheres dazu in den Abschnitten 8.4.1 und A.7.) Robuster ist der Ansatz, jedes nichtatomare

Teilproblem weiter zu zerlegen. Auf diesem Prinzip beruhen die bereits erwähnten Baumeinbettungsalgorithmen (LEIGHTON ET AL., 1989; RANADE, 1994) und Prozessorfarmalgorithmen wie (UMLAND UND VOLLMAR, 1992). Dieser Ansatz ist allerdings nur sinnvoll, wenn atomare Teilprobleme so grobgranular sind, daß sich der Aufwand für Erzeugen, Kommunizieren und Speichern der Teilprobleme lohnt.

Als Kompromiß zwischen diesen beiden extremen Ansätzen bietet es sich an, die Korngröße künstlich zu erhöhen, indem die maximale Tiefe, bis zu der gespalten wird, begrenzt wird. Selbst bei einem relativ regelmäßigen Problem wie dem 15-Puzzle muß aber immer noch eine sehr große Anzahl von Teilproblemen erzeugt werden, um eine ausreichende Lastverteilung zu ermöglichen (REINEFELD, 1994).[21] In Kapitel 5.3 wird untersucht, warum dies so ist, und in Kapitel 8 werden Algorithmen entwickelt, die die Erzeugung einer sehr großen Zahl von Teilproblemen verkraften, weil diese weder gespeichert noch transportiert werden müssen.

2.4.2 Empfängerveranlaßtes Spalten

```
process Worker                                   (* usually running on each PE *)
var P, P' : Subproblem
Get initial subproblem P from load balancer
while no global termination yet do
    if T(P) = 0 then
        get new work from load balancer
    while T(P) ≠ 0 do
        if there is a load request then
            (P, P') := split(P)
            send P' to requesting process
        P := work(P, Δt)
```

Abbildung 2.6: Generischer Algorithmus für empfängerveranlaßtes Spalten.

Das oben beschriebene Dilemma zwischen schlechter Lastverteilung und übermäßig feinkörnigem Spalten der Last läßt sich durchbrechen, indem Probleme adaptiv nur dann aufgespalten werden, wenn Last benötigt wird. Abbildung 2.6 beschreibt einen generischen Algorithmus für *empfängerveranlaßtes Spalten*, der die meisten bekannten adaptiven Lastverteilungsalgorithmen als Spezialfall enthält. Im allgemeinen läuft auf jedem PE ein[22] Arbeitsprozeß, der vom Lastverteiler ein anfängliches Teilproblem zugeteilt bekommt. In der Literatur wird meist P_{root} PE 0 zugeteilt und alle anderen PEs erhalten zunächst ein leeres Teilproblem. Bessere Verfahren werden in Abschnitt 8.4.1 diskutiert.

Solange ein Arbeitsprozeß ein nichtleeres Teilproblem hat, wird dieses sequentiell abgearbeitet. Arbeitslose PEs wenden sich an einen Lastverteilungsalgorithmus, dessen Aufgabe es

[21] Bessere Lösungen sind möglich, wenn eine gute Schätzfunktion für die Größe von Teilproblemen zur Verfügung steht. Dann kann die weitere Aufspaltung gestoppt werden, sobald eine bestimmte Mindestgranularität unterschritten wird.

[22] In Abschnitt 6.4 wird diskutiert, wann es Sinn macht, jedem PE mehrere Teilprobleme zuzuordnen.

ist, einen „Spenderprozeß“ zu finden, dessen Teilproblem aufgespalten wird, so daß ein Teil an den Arbeitslosen weitergegeben werden kann. Um auf Anforderungen zum Aufspalten reagieren zu können, wird die sequentielle Bearbeitung periodisch unterbrochen. [23)]

Entscheidend für die Effizienz des Gesamtalgorithmus ist die Strategie, mit der Anforderungen vermittelt werden. Naheliegend ist die Idee, daß arbeitslose PEs Anforderungen unmittelbar an benachbarte PEs senden. Dies führt aber dazu, daß sich um PEs mit großen Teilproblemen Agglomerate oder „Klumpen“ von arbeitenden PEs bilden, die das große Teilproblem abschirmen und eine weitere direkte Aufspaltung unmöglich machen. Arbeitslose PEs am Rande dieser Agglomerate erhalten nur vielfach aufgespaltene kleine Teilprobleme, deren Transport sich oft nicht lohnt. In RAO UND KUMAR (1987b) wird gezeigt, daß bei linearen Gittern exponentiell viele Aufspaltungen nötig werden. Bei höherdimensionalen Gittern ist die Situation nur wenig besser, und selbst bei Hyperwürfeln ist diese Agglomeration ein Problem, wenn h nicht nahe bei $\log n$ liegt. Nur auf diese negativen Ergebnisse für den Spezialfall der Nachbarschaftskommunikation kann auch die Aussage in REINEFELD (1994) bezogen werden, daß empfängerveranlaßtes Spalten ein schlecht skalierbares Lastverteilungsverfahren ist.

Empfängerveranlaßtes Spalten funktioniert um so besser, je gleichmäßiger Anforderungen auf die vorhandenen Teilprobleme verteilt werden. Eine einfache Methode, dies zu erzwingen, besteht darin, einen globalen Zähler z zu verwalten, der bestimmt, welches PE die nächste Anforderung erhält (KUMAR ET AL., 1994). Nach jeder Anforderung wird z auf $z+1 \bmod n$ gesetzt. Leider führt eine zentrale Verwaltung des Zählers zu einem Engpaß. Mit Hilfe einer in Abschnitt 5.2 hergeleiteten unteren Schranke läßt sich leicht zeigen, daß $T_{\text{par}} \in \Omega(hn)$. Dieser Engpaß wird durch die Verwaltung lokaler Zähler zwar beseitigt, aber dadurch können sich wieder erhebliche Ungleichverteilungen der Anfragen ergeben. Globale Zähler lassen sich jedoch durch eine geschicktere parallele Implementierung des Zählers so weit verbessern, daß die Verwaltung des Zählers nur noch einen Term $O(hT_{\text{coll}})$ zu parallelen Ausführungszeit beisteuert.[24)]

Selbst bei Verwaltung eines globalen Zählers werden viele Anforderungen an arbeitslose PEs gerichtet. Dies läßt sich vermeiden, indem in separaten Lastverteilungsphasen arbeitslose und beschäftigte PEs identifiziert und einander kollisionsfrei zugeordnet werden. Diese Operation ist mit Hilfe weniger Präfixsummen- und Datentransportoperationen möglich und asymptotisch genauso effizient wie globales Zählen. Da arbeitslose PEs bis zur nächsten Lastverteilungsphase warten müssen, wird dieses Verfahren allerdings hauptsächlich auf SIMD-Rechnern eingesetzt, bei denen ohnehin synchrone Lastverteilungsphasen nötig sind und die andererseits Präfixsummen meist effizient berechnen können (CHRISTMAN, 1983; HILLIS, 1985; DEHNE ET AL., 1990a; POWLEY ET AL., 1993; KARYPIS UND KUMAR, 1992, 1994; SANDERS, 1993, 1994d).

Die oben beschriebenen Lastverteilungsstrategien sind entweder ineffizient oder relativ kompliziert. Dies gilt nicht für den letzten hier vorzustellenden Algorithmus. Beim *zufälligen Anfragen* (*Random polling*) schickt ein arbeitsloses PE Lastanforderungen an zufällig ausgewählte PEs (gleichverteilt und unabhängig von den anderen PEs). Dieses Verfahren ist so einfach, daß es wohl mehrfach entdeckt wurde. Schon in FINKEL UND MANBER (1987) findet sich ein ähnlicher Ansatz. In MONIEN ET AL. (1990) wird zufälliges Anfragen für Spielbaumsuche angewandt. Für vielfädige Berechnungen wird es in BLUMOFE UND LEISERSON (1994) benutzt. Eine Variante für SIMD findet sich in SANDERS (1993, 1994d), die

[23)] Alternativ kann eine Spaltanforderung ein Unterbrechungssignal auslösen.

[24)] Bisherige Implementierungen (KUMAR UND ANANTH, 1991) weisen darauf hin, daß der konstante Faktor, der sich hinter dem Zählaufwand verbirgt, doch recht hoch ist. Dies mag sich allerdings durch modernere Rechnerarchitekturen oder effizientere (randomisierte) Algorithmen (wie in SHAVIT UND ZEMACH (1994)) ändern.

in SANDERS (1994b), HOPP UND SANDERS (1995) und Abschnitt 6.1 weiter verfeinert wird. Die Darstellung hier bezieht sich auf KUMAR UND ANANTH (1991), wo es sich empirisch als das beste bekannte Verfahren für parallele Tiefensuche herausstellt. Außerdem wird dort nachgewiesen, daß für die mittlere Ausführungszeit gilt[25)]

$$\mathbf{E}T_{\text{par}} \in \mathrm{O}\left(\frac{T_{\text{seq}}}{n} + hd\log n\right) \quad .$$

Der zweite Term ist mit $hd\log$ um einen logarithmischen Faktor schlechter als für eine ebenfalls untersuchte effiziente Implementierung des globalen Zählens. Diese Diskrepanz zwischen experimentellem und analytischem Ergebnis war ein wichtiger Ausgangspunkt für diese Arbeit. Tatsächlich stellt sich in Kapitel 6 heraus, daß zufälliges Anfragen im Mittel und mit hoher Wahrscheinlichkeit um eben diesen logarithmischen Faktor besser ist, als die obige Abschätzung. In den Kapiteln 7 und 8 zeigt sich außerdem, daß eine Verfeinerung von zufälligem Anfragen eine Verbesserung um einen weiteren logarithmischen Faktor ermöglicht.

2.5 Zusammenfassung

In diesem Kapitel wurde der Grundstein für Entwurf und Analyse von Lastverteilungsalgorithmen für baumförmige Berechnungen gelegt, die eine gute Abstraktion vieler praktischer Probleme und theoretischer Modelle sind. Ziel der Analysen ist es, eine Schranke für die parallele Ausführungszeit T_{par} herzuleiten, aus der sich andere Größen wie Beschleunigung, Effizienz oder Skalierbarkeit ablesen lassen. Da es sich hauptsächlich um randomisierte Algorithmen handelt, werden die Schranken probabilistische Aussagen im $\tilde{\mathrm{O}}$-Kalkül sein. Was die Maschine betrifft, so wird T_{par} hauptsächlich von den Parametern n für Prozessoranzahl, d für Netzwerkdurchmesser und b für Bisektionsbreite abhängen. Trotzdem wird wenig von maschinenspezifischen Eigenheiten die Rede sein, da diese Parameter meist indirekt, vermittels des Aufwands für Datentransportprobleme und kollektive Operationen, ins Spiel kommen. Die wichtigsten Parameter der baumförmigen Berechnungen sind die sequentielle Ausführungszeit T_{seq}, die Spalttiefe h, die Problemgranularität T_{atomic}, der Spaltaufwand T_{split} und die Länge l einer Teilproblembeschreibung. Zur Parallelisierung baumförmiger Berechnungen sind vor allem adaptive Algorithmen geeignet, die Teilprobleme nur dann spalten, wenn ein konkreter Bedarf dafür besteht. Da Teilproblemgrößen nicht vorhersagbar sind, müssen Lastverteilungsalgorithmen darauf bedacht sein, bei minimalem Kommunikationsaufwand alle vorhandenen Teilprobleme ausreichend oft aufzuspalten.

Ergebnisse

Das Konzept der baumförmigen Berechnung ist hinreichend verschieden von existierenden Modellen, um es als neu zu bezeichnen und zeichnet sich durch eine Kombination von Allgemeinheit, Einfachheit und Praxisnähe aus. Die Abschnitte 2.1 und 2.2 sind hauptsächlich Aufbereitungen existierender Arbeiten. Der Begriff des $\tilde{\mathrm{O}}$-Kalküls ist allerdings in der Literatur bisher weniger genau definiert worden.

[25)] Für $h \in \mathrm{O}(\log n)$ entsprechend dem α-Spaltmodell aus Abschnitt 2.3.5, $\{l, T_{\text{atomic}}, T_{\text{split}}\} \subseteq \mathrm{O}(1)$ und hinreichend großes b.

Kapitel 3

Analyse randomisierter Algorithmen

In Abschnitt 3.1 werden zunächst einige Abschätzungen für die Wahrscheinlichkeit von Ereignissen eingeführt. Darauf aufbauend wird dann ein Grundstock eines „Werkzeugkastens“ für die Analyse randomisierter Algorithmen konstruiert. Begonnen wird in Abschnitt 3.2 mit Aussagen über Summen unabhängiger 0/1-Zufallsvariablen. Herzstück des Kapitels ist Abschnitt 3.3, in dem Rechenregeln für den $\tilde{O}$-Kalkül in Anlehnung an den deterministischen O-Kalkül hergeleitet werden. In Abschnitt 3.4 stellt sich heraus, daß Aussagen über das Verhalten mit hoher Wahrscheinlichkeit in vielen Fällen stärker sind, als solche über das Verhalten im Mittel. Abschnitt 3.5 zeigt eine Anwendung der hier erarbeiteten Werkzeuge auf die Analyse der häufig benutzten Technik des zufälligen Verteilens von „Bällen“ (lies Nachrichten, Datenelemente) auf „Kisten“ (lies Prozessoren, Feldelemente). Schließlich gibt Abschnitt 3.6 eine Zusammenfassung und diskutiert weitere Ausbaumöglichkeiten.

3.1 Wahrscheinlichkeitsschranken

Ein grundlegendes Hilfsmittel für den Nachweis, daß eine Aussage mit hoher Wahrscheinlichkeit gilt, sind obere Schranken für die Wahrscheinlichkeit von Ereignissen. Besonders einfach aber nützlich ist die in jedem Analysiskurs auftretende Abschätzung

$$\left(1-\frac{1}{n}\right)^m \leq e^{-\frac{m}{n}} \quad . \tag{3.1}$$

Die in Formeln für Wahrscheinlichkeiten häufig auftretenden Binomialkoeffizienten lassen sich oft schon mit

$$\binom{n}{k} \leq \left(\frac{ne}{k}\right)^k \tag{3.2}$$

hinreichend genau abschätzen (z.B. MOTWANI UND RAGHAVAN (1995) Proposition B.2). Diese Formel beruht auf der Stirlingapproximation der Fakultätsfunktion.

Nur folgende eigentlich triviale Aussage ist i.allg. anwendbar, wenn es darum geht, die Wahrscheinlichkeit abzuschätzen, daß das Maximum n identisch verteilter Zufallsvariablen X_i ($i \in \mathbb{N}_n$) eine Schranke m überschreitet:

$$\mathbf{P}\left[\max_{i<n} X_i > m\right] = \mathbf{P}\left[\bigvee_{i<n} X_i > m\right] \leq n\mathbf{P}[X_j > m] \text{ für beliebiges } j. \tag{3.3}$$

Ein wichtiger Vorteil dieser Abschätzung ist, daß sie beliebige Abhängigkeiten zwischen den X_i zuläßt.

Besonders wichtig sind Schranken für die Wahrscheinlichkeit großer Abweichungen vom Erwartungswert. So gelten für die Binomialverteilung die folgenden auf CHERNOFF (1952) zurückgehenden Schranken:

Satz 3.1. *Seien X_i ($i \in \mathbb{N}_n$) unabhängige 0/1-Zufallsvariablen. Sei $X = \sum_{i<n} X_i$ und $q \leq \mathbf{P}[X_i = 1] \leq p$. Dann gilt (*LEIGHTON*, 1992;* RAJASEKARAN*, 1992)*

$$\forall \beta > 1 : \mathbf{P}[X \geq \beta pn] \leq e^{(1-1/\beta-\ln\beta)\beta pn} \tag{3.4}$$

$$\forall \varepsilon \in (0,1) : \mathbf{P}[X \geq (1+\varepsilon)pn] \leq e^{-\varepsilon^2 pn/2} \tag{3.5}$$

$$\forall \varepsilon \in (0,1) : \mathbf{P}[X \leq (1-\varepsilon)qn] \leq e^{-\varepsilon^2 qn/3} \tag{3.6}$$

Relation 3.4 wird hier nur in der abgeschwächten Form

$$\mathbf{P}[X \geq \beta pn] \leq e^{(1-\ln\beta)\beta pn} \tag{3.7}$$

benötigt.

Allgemeinerer Formulierungen findet man z.B. in ALON UND SPENCER (1992). Chernoff-Schranken vereinigen die Vorteile einfacher Anwendbarkeit, recht großer Allgemeinheit und genauer Aussagen in sich. (Noch genauere Aussagen wie in WORSCH (1994) sind komplizierter und ergeben hier keine Verbesserung der Gesamtergebnisse.)

Relativ unbekannt ist, daß sich die obigen Chernoff-Schranken auch auf die hypergeometrische Verteilung anwenden lassen, da diese durch die Binomialverteilung abgeschätzt werden kann.[1)] Besonders anschaulich ist dabei die Formulierung als Urnenmodell. Während die Binomialverteilung die Anzahl Erfolge bei n-maligem Ziehen aus einer Urne mit Zurücklegen angibt, beschreibt die hypergeometrische Verteilung die Anzahl der Erfolge beim Ziehen ohne Zurücklegen. Die folgende Aussage wurde leicht umformuliert, um sich reibungsloser in die hier verwendeten Bezeichner einzufügen.

Lemma 3.2 (Satz 6.81 in JOHNSON ET AL. (1989)**).** *Gegeben sei eine Urne mit m Kugeln, von denen pm weiß sind. Sei X die Anzahl weißer Kugeln, die bei k-maligem Ziehen ohne Zurücklegen gezogen werden. Sei Y die Anzahl weißer Kugeln, die bei k-maligem Ziehen mit Zurücklegen gezogen werden. Dann gilt*

$$\mathbf{P}[X > a] \leq \mathbf{P}[Y > a] \text{ falls } p \leq \frac{a}{(k-1)} \cdot \frac{m}{m+1} \quad .$$

Wird das Lemma für den Fall einer positiven Abweichung vom Erwartungswert umformuliert, so zeigt sich, daß die Bedingung immer erfüllt ist:

[1)] Ich möchte Bernhard Klar vom Institut für mathematische Statistik für diesen Hinweis danken.

Satz 3.3. *Gegeben sei eine Urne mit m Kugeln von denen pm weiß sind. Sei X die Anzahl weißer Kugeln, die bei k-maligem Ziehen ohne Zurücklegen gezogen werden. Sei Y die Anzahl weißer Kugeln, die bei k-maligem Ziehen mit Zurücklegen gezogen werden. Sei* $\varepsilon \geq 0$ *beliebig. Dann gilt*

$$\mathbf{P}[X > (1+\varepsilon)pk] \leq \mathbf{P}[Y > (1+\varepsilon)pk] \quad .$$

Beweis. Aus Lemma 3.2 folgt für $a = (1+\varepsilon)pk$ die Behauptung falls

$$p \leq \frac{(1+\varepsilon)pk}{(k-1)} \frac{m}{m+1} \quad .$$

Durch Kürzen und Ausmultiplizieren ergibt sich die Bedingung $km+k-m-1 \leq km+\varepsilon km$ oder $k-m-1 \leq \varepsilon km$. Wegen $k \leq m$ ist dies immer gegeben. □

Leider treten in der Algorithmenanalyse oft komplexe Abhängigkeiten zwischen Zufallsvariablen auf. Eine der wenigen Aussagen, die auch dann noch eine scharfe Konzentration um den Erwartungswert voraussagt ist *Azumas Ungleichung*:

Satz 3.4 (Satz (4.16) in MOTWANI UND RAGHAVAN (1995)**).** *Bilden die Zufallsvariablen* X_k *(*$k \in \mathbb{N}_{m+1}$*) eine Martingalfolge mit der Eigenschaft* $|X_{k+1} - X_k| \leq c_k$*, dann gilt für beliebige* $\lambda > 0$

$$\mathbf{P}[|X_m - X_0| \geq \lambda] \leq 2e^{-\frac{\lambda^2}{2\sum_{k<m} c_k{}^2}} \quad .$$

Auf die Definition des Begriffs Martingal sei an dieser Stelle verzichtet, da dieser nur in Abschnitt 6.2.3 in einer sehr speziellen Form auftaucht.

3.2 0/1-Zufallsvariablen

Die Chernoff-Schranken aus Satz 3.1 liefern Schranken für die Wahrscheinlichkeit, daß Summen von 0/1-Zufallsvariablen (auch *Indikatorzufallsvariablen genannt*) eine gegebene Abweichung vom Erwartungswert aufweisen. Für eine hinreichend große Zahl von Variablen läßt sich dies leicht in die Aussage ummünzen, daß asymptotisch relevante Abweichungen vom Mittelwert mit hoher Wahrscheinlichkeit nicht auftreten. „Hinreichend groß“ ist dabei im Verhältnis zum Parameter *n* zu verstehen, bezüglich dem hohe Wahrscheinlichkeit definiert ist.

Satz 3.5. *Seien* Y_i *(*$i \in \mathbb{N}_m$*) unabhängige Zufallsvariablen mit* $\mathbf{Bild}\, Y_i = \{0,1\}$ *und* $\mathbf{P}[X_i = 1] \leq p$*. Dann gilt*[2)]

$$Y := \sum_{i<m} Y_i \in \tilde{\mathrm{O}}(\max(mp, \log n)) \quad .$$

Beweis. Es genügt zu zeigen, daß $Y \in \tilde{\mathrm{O}}(mp + \ln n)$. Für beliebiges $\beta > 0$ wähle $c = \max(\beta, e^2)$. Dann gilt mit Hilfe der Chernoff-Schranke (3.7),

$$\begin{aligned}\mathbf{P}[Y > c\max(mp, \ln n)] &= \mathbf{P}\left[Y > mp \cdot \max\left(c, \frac{c\ln n}{mp}\right)\right] \\ &\leq e^{\left(1 - \ln\max\left(c, \frac{c\ln n}{mp}\right)\right)} \max\left(c, \frac{c\ln n}{mp}\right) mp\end{aligned}$$

[2)] Das Auftauchen von *n* in der Formel rührt aus der Definition von $\tilde{\mathrm{O}}$.

Durch die Abschätzungen $\max\left(c, \frac{c\ln n}{mp}\right) \geq \frac{c\ln n}{mp}$, $\max\left(c, \frac{c\ln n}{mp}\right) \geq c$
und $\ln c \geq \ln e^2 = 2$ vereinfacht sich dies zu

$$\leq e^{(1-\ln c)\, c\ln n} \leq e^{-c\ln n} \leq n^{-\beta} \quad .$$

□

Für $mp \notin \Omega(\log n)$ kann Satz 3.5 Abweichungen bis zu $\log n$ vom Mittelwert nicht ausschließen. Genauere Aussagen erfordern etwas aufwendigere Rechnungen. Dies sei am Beispiel des wichtigen Falles $mp \in \mathrm{O}(1)$ vorgeführt.

Satz 3.6. *Seien Y_i ($i \in \mathbb{N}_m$) unabhängige Zufallsvariablen mit* **Bild** $Y_i = \{0,1\}$, $mp \in \mathrm{O}(1)$ *und* $\mathbf{P}[Y_i = 1] \leq p$*). Dann gilt*

$$Y := \sum_{i<m} Y_i \in \tilde{\mathrm{O}}\left(\frac{\log n}{\log\log n}\right) \quad .$$

Beweis. Der Ansatz ist wie im Beweis von Satz 3.5. Nur wird $c \geq \max\left(\frac{\beta}{1-1/e}, emp\right)$ gewählt.

$$\begin{aligned}
\mathbf{P}\left[Y > c\frac{\ln n}{\ln\ln n}\right] &= \mathbf{P}\left[Y > mp\cdot\frac{c\ln n}{mp\ln\ln n}\right] \\
&\leq e^{\left(1-\ln\frac{c\ln n}{mp\ln\ln n}\right)c\frac{\ln n}{\ln\ln n}} \\
&= n^{-\frac{c}{\ln\ln n}\left(-1+\ln\frac{c}{mp}+\ln\ln n-\ln\ln\ln n\right)} \\
&= n^{-c\left(1+\frac{\ln\frac{c}{mp}-1}{\ln\ln n}-\frac{\ln\ln\ln n}{\ln\ln n}\right)} \\
&\leq n^{-c\left(1-\frac{\ln\ln\ln n}{\ln\ln n}\right)} \text{ wegen } \frac{c}{mp} \geq e
\end{aligned}$$

Durch Nullsetzen der Ableitung und Randwertbetrachtung läßt sich außerdem zeigen, daß $\frac{\ln\ln\ln n}{\ln\ln n} \leq \frac{1}{e}$ für $n \geq e^e$.

$$\mathbf{P}\left[Y > c\frac{\ln n}{\ln\ln n}\right] \leq n^{-c\left(1-\frac{1}{e}\right)} \leq n^{-\beta}$$

□

Falls p nicht nur nach oben, sondern auch nach unten beschränkt ist, ermöglichen Chernoff-Schranken auch Aussagen über die absolute Abweichung der Summe von ihrem Erwartungswert. Hier wird nur der einfache Fall einer genau bestimmten Wahrscheinlichkeit benötigt.

Satz 3.7. *Gegeben seien unabhängige Zufallsvariablen Y_i ($i \in \mathbb{N}_m$) mit* **Bild** $Y_i = \{0,1\}$ *und* $\mathbf{P}[Y_i = 1] = p$*. Sei außerdem $mp \in \Omega(\log n)$. Dann gilt*

$$\left|mp - \sum_{i<m} Y_i\right| \in \tilde{\mathrm{O}}\left(\sqrt{mp\log n}\right) \quad .$$

Beweis. Falls $mp \in O(\log n)$ folgt die Aussage aus Satz 3.5. Sonst genügt es zu zeigen, daß $|mp - \sum_{i<m} Y_i| \in \tilde{O}\left(\sqrt{mp \ln n}\right)$. Für $\beta > 0$ seien c und n_0 so gewählt, daß $n^{-c/3} + n^{-c/2} \leq n^{-\beta}$ für $n \geq n_0$. (z.B. $c = 3\beta + 1$, $n_0 = 6$). Es gilt

$$\begin{aligned} P_{\text{diff}} &:= \mathbf{P}\left[\left|mp - \sum_{i<m} Y_i\right| > c\sqrt{mp \ln n}\right] \\ &= \mathbf{P}\left[\sum_{i<m} Y_i > mp\left(1 + c\sqrt{\frac{\ln n}{mp}}\right) \vee \sum_{i<m} Y_i < mp\left(1 - c\sqrt{\frac{\ln n}{mp}}\right)\right] \end{aligned}$$

Wegen $mp \notin O(\ln n)$ gilt $c\sqrt{\frac{\ln n}{mp}} < 1$ für hinreichend große n und die Chernoff-Schranken (3.5) und (3.6) können angewandt werden.

$$P_{\text{diff}} \leq e^{-\frac{cmp\ln n}{3mp}} + e^{-\frac{cmp\ln n}{2mp}} = n^{-\frac{c}{3}} + n^{-\frac{c}{2}} \leq n^{-\beta} \quad .$$

□

Insbesondere sind die Abweichungen für $mp \notin O(\log n)$ asymptotisch vernachlässigbar klein. (Also $\lim_{n\to\infty} \sum_{i<m} Y_i / mp = 1$).

3.3 Rechenregeln für den $\tilde{O}$-Kalkül

Ziel dieses Abschnittes ist es, Analogien zwischen dem altbekannten deterministischen O-Kalkül und dem $\tilde{O}$-Kalkül aufzuzeigen. Ein logischer Startpunkt dafür ist die recht einfache Beobachtung, daß im deterministischen Sinne wahre Aussagen auch im probabilistischen Sinne wahr sind. Formulierung und Beweis einer entsprechenden Regel sind nicht sehr tiefsinnig, liefern aber einen ersten Konsistenztest der Definition des Verhaltens mit hoher Wahrscheinlichkeit.

Satz 3.8. *Aus $X \in O(f(n))$ folgt $X \in \tilde{O}(f(n))$*

Beweis. $X \in O(f(n))$ ist als $\max\{\textbf{Bild}\, X\} \in O(f(n))$ zu lesen (siehe auch Abschnitt 2.1). Damit folgt aus der Definition von $O(\cdot)$, daß $\mathbf{P}[X > cf(n)] = 0 \leq n^{-\beta}$ für geeignetes c, hinreichend große n und beliebiges $\beta > 0$. □

Komplizierter, aber dafür sehr mächtig ist die folgende Aussage, die Schlüsse über das Verhalten einer Zufallsvariablen erlaubt, deren Wert von einer anderen Zufallsvariablen abhängt.

Satz 3.9. *Sei $X(Y)$ eine Zufallsvariable, die vom Wert einer zweiten Zufallsvariablen Y abhängt (**Bild** $X \subseteq \mathbb{R}_*$, **Bild** $Y \subseteq \mathbb{R}_*$). Gilt $Y \in \tilde{O}(g(n))$, $X(Y) \in \tilde{O}(f(Y,n))$ (wenn Y als Parameter angesehen wird, der eine nicht näher bestimmte Zahl repräsentiert) und*

$$\forall c \in \mathbb{R}_* : \exists n_0, c' \in \mathbb{R}_* : \forall n \geq n_0 : f(cg(n), n) \leq c' f(g(n), n),$$

dann folgt

$$X(Y) \in \tilde{O}(f(g(n), n)) \quad .$$

Beweis. Für beliebiges $\beta > 0$ seien c, c' und n_0 so gewählt, daß für $n \geq n_0$ gilt $\mathbf{P}[X(g(n)) > cf(g(n),n)] \leq n^{-\beta-1}$, $\mathbf{P}[Y > c'g(n)] \leq n^{-\beta-1}$ und $f(cg(n),n) \leq c'f(g(n),n)$. Dann gilt

$$\begin{aligned}\mathbf{P}[X(Y) \geq cf(g(n),n)] &\leq \mathbf{P}\left[X(g(n)) \geq cf(g(n),n) \vee Y \geq c'g(n)\right] \\ &\leq 2n^{-\beta-1} \leq n^{-\beta}\end{aligned}$$

für $n \geq \max(n_0, 2)$. □

Die Voraussetzung $f(cg(n),n) \leq c'f(g(n),n)$ ist dabei keine Einschränkung, die vom probabilistischen Ansatz herrührt; sie muß auch für eine entsprechende deterministische Regel gemacht werden.

Die üblichen O-Kalkülregeln für Summe, Maximum und Produkt folgen unmittelbar aus der obigen *Kettenregel*, indem für g die identische Abbildung und für f Summe, Maximum bzw. Produkt eingesetzt werden.

Korollar 3.10. *Sei $X \in \tilde{\mathrm{O}}(f(n))$ und $Y \in \tilde{\mathrm{O}}(g(n))$ dann gilt*

$$X \otimes Y \in \tilde{\mathrm{O}}(f(n) \otimes g(n)) \text{ für } \otimes \in \{+, \max, \cdot\} \quad .$$

Die arithmetischen Regeln dürfen mehrfach hintereinander angewandt werden, und es ergeben sich entsprechende Regeln für Summe, Produkt und Maximum mehrerer Variablen. Vorsicht ist aber geboten, wenn die Anzahl der Operanden von n abhängt. Dann können die involvierten konstanten Faktoren so schnell steigen, daß sie das asymptotische Verhalten mitbestimmen. Hier ein Beispiel mit deterministischen Aussagen: Sei $x_i = 2^i$. Jeder der Werte für sich betrachtet ist eine Konstante und es gilt $x_i \in \mathrm{O}(1)$. Aber $\sum_{i<n} x_i = 2^n - 1 \notin \mathrm{O}(n \cdot 1)$.

Zumindest für Summe und Maximum läßt sich dieses Problem aber lösen, wenn etwas über den Zusammenhang der involvierten Konstanten bekannt ist. Insbesondere ist dies der Fall, wenn alle betrachteten Variablen einer gemeinsamen Schranke unterliegen.

Satz 3.11. *Gegeben seien Zufallsvariablen X_i ($i \in \mathbb{N}_m$), die durch eine Zufallsvariable X beschränkt sind, also*

$$\forall i \in \mathbb{N}_m : \forall x \in \mathbb{R}_* : \mathbf{P}[X_i > x] \leq \mathbf{P}[X > x] \quad .$$

Falls m polynomiell in n ist, folgt aus $X \in \tilde{\mathrm{O}}(f(n))$

$$\sum_{i<m} X_i \quad \in \quad \tilde{\mathrm{O}}(mf(n)) \qquad \text{und} \tag{3.8}$$

$$\max_{i<m} X_i \quad \in \quad \tilde{\mathrm{O}}(f(n)) \quad . \tag{3.9}$$

Beweis. Für beliebiges $\beta > 0$ seien c und n_0 so gewählt, daß

$$\mathbf{P}[X_i > cf(n)] \leq \mathbf{P}[X > cf(n)] \leq n^{-\beta - \frac{\log m}{\log n}} \text{ für } n \geq n_0 \quad .$$

Dies ist möglich, weil m polynomiell in n ist und damit $\frac{\log m}{\log n}$ durch eine Konstante beschränkt ist. Dann gilt unter Anwendung der Abschätzung 3.3

$$\mathbf{P}\left[\sum_{i<m} X_i \geq cmf(n)\right] \leq \mathbf{P}\left[\bigvee_{i<m} X_i \geq cf(n)\right] \leq mn^{-\beta - \frac{\log m}{\log n}} = n^{-\beta}$$

für $n \geq n_0$. Der Beweis für die Maximumbildung verläuft analog. □

3.4 Zusammenhang mit Erwartungswerten

Im allgemeinen sind Ergebnisse über das Verhalten einer Zufallsvariablen mit hoher Wahrscheinlichkeit einerseits und über ihren Erwartungswert andererseits nicht vergleichbar. Ist z.B. $\mathbf{P}[X = 1] = 1 - 2^{-n}$ und $\mathbf{P}[X = 4^n] = 2^{-n}$ so gilt $X \in \tilde{\mathrm{O}}(1)$ aber $\mathbf{E}X > 2^n$. In den meisten tatsächlich auftretenden Fällen ist eine Aussage im $\tilde{\mathrm{O}}$-Kalkül aber stärker als eine Aussage über den Erwartungswert. Der folgende Satz formuliert eine hinreichende Bedingung dafür, die für die meisten hier betrachteten Verteilungen ausreicht.

Satz 3.12. *Gilt für eine Zufallsvariable mit nichtnegativen Werten $X \in \tilde{\mathrm{O}}(f(n))$ und gibt es ein k, so daß $X \in \mathrm{O}\big(n^k f(n)\big)$, so folgt $\mathbf{E}X \in \mathrm{O}(f(n))$.*

Beweis. Seien c, c' und k so gewählt, daß $\mathbf{P}[X > cf(n)] \leq n^{-k}$ für $n \geq n_0$ und $X \leq c'n^k f(n)$ für $n \geq n_0'$. Sei $\varphi(x)$ die Wahrscheinlichkeitsdichtefunktion von X. Dann gilt

$$\mathbf{E}X = \int_0^\infty x\varphi(x)dx$$

Der Integrationsintervall wird nun in drei Teile zerlegt. Einen, der im Bereich der gewünschten Schranke für den Erwartungswert liegt, einen der darüber liegt aber eine geringe Wahrscheinlichkeitsdichte aufweist, und einen der über der oberen Schranke für X liegt und deshalb keinen Beitrag zur Gesamtwahrscheinlichkeit liefert.

$$\begin{aligned}
\mathbf{E}X &= \int_0^{cf(n)} x\varphi(x)dx + \int_{cf(n)}^{c'n^k f(n)} x\varphi(x)dx + \int_{c'n^k f(n)}^{\infty} x\varphi(x)dx \\
&\leq cf(n) \int_0^{cf(n)} \varphi(x)dx + c'n^k f(n) \int_{cf(n)}^{c'n^k f(n)} \varphi(x)dx + \int_{c'n^k f(n)}^{\infty} x \cdot 0 \cdot dx \\
&= cf(n)\mathbf{P}[X \leq cf(n)] + c'x^k f(n)\mathbf{P}[X > cf(n)] + 0 \\
&\leq cf(n) \cdot 1 + c'n^k f(n)n^{-k} = (c + c')f(n)
\end{aligned}$$

für $n \geq \max(n_0, n_0')$. Unterliegt X einer diskreten Wahrscheinlichkeitsverteilung, genügt es, in der obigen Rechnung die Integralzeichen als Summenzeichen zu lesen. □

Eine leichte Umformulierung dieses Satzes erlaubt es, die in den folgenden Kapiteln hergeleiteten Aussagen über die Ausführungszeit von parallelen Algorithmen auch als Aussagen über Erwartungswerte zu lesen:

Satz 3.13. *Gilt für eine Zufallsvariable $X \preceq g(n) + \tilde{\mathrm{O}}(f(n))$ und gibt es ein k, so daß $X \preceq g(n) + \mathrm{O}\big(n^k f(n)\big)$, so folgt $\mathbf{E}X \preceq g(n) + \mathrm{O}(f(n))$.*

Beweis. Wende Satz 3.12 auf die Zufallsvariable $\max(X - g(n), 0)$ an. □

Da alle untersuchten Lastverteilungsalgorithmen so formuliert sind, daß ihre parallele Ausführungszeit auf jeden Fall in $O\left(T_{\text{seq}}\right)$ bleibt, ist die Bedingung für die Anwendung von Satz 3.13 immer erfüllt. Ergebnisse wie $T_{\text{par}} \preceq (1+\varepsilon)\frac{T_{\text{seq}}}{n} + \tilde{O}(f(n))$ implizieren also jeweils $\mathbf{E}T_{\text{par}} \preceq (1+\varepsilon)\frac{T_{\text{seq}}}{n} + O(f(n))$, ohne daß dies im folgenden gesondert vermerkt wird.

3.5 Zufälliges Zuordnen

Viele randomisierte Algorithmen beruhen auf dem zufälligen Verteilen von Nachrichten oder Lasteinheiten auf PEs. Dafür gibt es ein einfaches mathematisches Modell, das sich häufig als nützlich erweist. Betrachtet wird ein System aus n „Kisten“ und m „Kugeln“. Die Kugeln werden unabhängig voneinander und gleichverteilt den Kisten zugeordnet.

Satz 3.14. *Seien X_i ($i \in \mathbb{N}_n$) die Anzahl Kugeln, die in Kiste i landen. Ferner sei m polynomiell in n. Dann gilt*

$$\max_{i<n} X_i \in \tilde{O}\left(\max\left(\frac{m}{n}, \log n\right)\right) \tag{3.10}$$

$$\max_{i<n} X_i \in \tilde{O}\left(\frac{\log n}{\log\log n}\right) \text{ falls } m \in O(n) \tag{3.11}$$

$$\max_{i<n} X_i - \min_{i<n} X_i \in \tilde{O}\left(\sqrt{\frac{m}{n}\log n}\right) \text{ falls } m \in \Omega(n \log n) \tag{3.12}$$

Beweis. Die 0/1-Zufallsvariable X_{ij} ($i \in \mathbb{N}_n$, $j \in \mathbb{N}_m$) sei definiert als $X_{ij} = 1$ gdw. „Kugel j trifft Kiste i“. Offenbar gilt $\sum_{j<m} X_{ij} = X_i$ und für festes i sind die X_{ij} unabhängig. Die Aussagen folgen nun ohne viel Rechnung aus einer Kombination der Maximumsregel (3.8) mit den Aussagen über Summen unabhängiger 0/1-Zufallsvariablen (Satz 3.5 und 3.6) und deren Abweichungen vom Erwartungswert (Satz 3.7). □

3.6 Zusammenfassung

Es wurde hier ein Satz von Hilfmitteln erarbeitet, die es ermöglichen, auf einfache Weise Schlüsse über probabilistische Algorithmen zu ziehen. Im Zentrum steht dabei der Begriff des Verhaltens mit hoher Wahrscheinlichkeit, der es erlaubt, komplexe Rechnungen in wiederverwendbaren Regeln abzukapseln. An die Stelle der direkten Rechnung (z.B. mit Chernoff-Schranken) treten einfache Aussagen über das asymptotische Verhalten von Summen von 0/1-Zufallsvariablen. Die aus dem O-Kalkül bekannten Rechenregeln lassen sich weitgehend übernehmen.

Dies macht die Analyse des Verhaltens mit hoher Wahrscheinlichkeit oft einfacher zu handhaben als das Rechnen mit Erwartungswerten. Dort gibt es zwar eine einfache Regel für die Addition von Zufallsvariablen. Schon bei der Multiplikation ist aber über abhängige Zufallsvariablen wenig Allgemeingültiges zu sagen. Die bei der Parallelverarbeitung besonders wichtige Maximierung schließlich ist beim direkten Rechnen mit Erwartungswerten nur schwer zu handhaben.

Die Aussagen sind so allgemein, daß sie nicht nur für die Beweise innerhalb dieser Arbeit, sondern auch für die Analyse vieler anderer Algorithmen geeignet sind. Zum Beispiel wird in SANDERS (1995c)

ein randomisierter Algorithmus zur Verwaltung paralleler Priority queues vorgestellt, dessen Analyse ausführlichen Gebrauch von den hier vorgestellten Hilfsmitteln macht.

Trotzdem ist an einigen Stellen eine weitere Verallgemeinerung möglich, auf die verzichtet wurde, um die Aussagen kompakt und übersichtlich zu halten. So lassen sich die in Abschnitt 3.2 beschriebenen Ergebnisse über Summen von 0/1-Zufallsvariablen für die in Abschnitt 3.1 erwähnten allgemeineren Chernoff-Schranken verallgemeinern. Aussagen über Summe und Maximum polynomiell vieler Zufallsvariablen wie in Satz 3.11 lassen sich auch dann treffen, wenn die Variablen keiner gemeinsamen Schranke unterliegen. Allerdings muß dann ausgeschlossen werden, daß die sich hinter dem $\tilde{O}$ verbergenden Konstanten von der Anzahl der betrachteten Variablen abhängen. Für die Summe unabhängiger Zufallsvariablen, kann auch die Beschränkung auf polynomiell viele Variablen fallengelassen werden. Der Erwartungswert einer Zufallsvariablen läßt sich oft sogar dann durch das Verhalten mit hoher Wahrscheinlichkeit beschränken, wenn der Wertebereich der Variablen unbeschränkt ist, aber einige zusätzliche Eigenschaften der zugrundeliegenden Verteilung bekannt sind. Eine solche Aussage findet sich z.B. in (SANDERS, 1994a).

Ergebnisse

Obwohl der Begriff des Verhaltens mit hoher Wahrscheinlichkeit in vielen Arbeiten eine Rolle spielt, wurden bisher keine systematischen Hilfsmittel zur einfachen Handhabung dieses Begriffs entwickelt. Der hier entwickelte Satz von „Werkzeugen“ hilft dieser Situation bis zu einem gewissen Grade ab. Die Analyse randomisierter Algorithmen kann oft in ähnlich übersichtlicher Weise formuliert werden, wie dies bisher nur im deterministischen O-Kalkül möglich war.

Kapitel 4

Basisalgorithmen

Der Entwurf sequentieller Algorithmen baut auf einer Anzahl Basisalgorithmen auf, über die seit längerer Zeit weitgehende Einigkeit besteht. Techniken wie verkettete Listen, Hashing, Sortieren und Zufallsgeneratoren werden in den meisten Lehrbüchern auf ähnliche Weise dargestellt, und ihre Wichtigkeit für den Entwurf komplexerer Algorithmen ist weitgehend unumstritten. Bei parallelen Algorithmen ist der Prozeß des Herauskristallisierens allgemein anerkannter Basisalgorithmen dagegen noch in vollem Gange.

Bei den in dieser Arbeit entwickelten Lastverteilungsalgorithmen werden Techniken verwendet, die wenig mit dem speziellen Problem zu tun haben und für eine Vielzahl von Anwendungen verwendbar sind. Einige dieser Basisalgorithmen werden hier weiterentwickelt und sind zu diesem Kapitel zusammengefaßt, um eine modulare Darstellung zu erreichen und um einen kleinen Beitrag zur Herausbildung eines Satzes von Grundbausteinen für parallele Algorithmen zu bilden.

In Abschnitt 4.1 werden die Vor- und Nachteile unterschiedlicher Terminierungserkennungsprotokolle diskutiert. Ein einfaches randomisiertes Verfahren zur näherungsweisen Beobachtung der globalen Summe verteilt vorliegender Werte wird in Abschnitt 4.2 eingeführt. Es zeichnet sich durch einen geringen Kommunikationsaufwand aus. Abschnitt 4.3 beschäftigt sich mit der parallelen Erzeugung von Zufallspermutationen.

4.1 Terminierungserkennung

Die scheinbar einfache Frage, wann alle PEs ihre Arbeit beendet haben, ist nicht immer leicht zu beantworten, insbesondere, wenn die Auslieferung einer Nachricht einen arbeitslosen Prozessor wieder aktivieren kann. Glücklicherweise kann bei dieser Fragestellung auf Arbeiten im Bereich der verteilten Systeme zurückgegriffen werden. Eine ausführliche Betrachtung des Problems der *Terminierungserkennung* findet sich in MATTERN (1987).

Leider arbeitet die Analyse in diesem Bereich fast ausschließlich mit dem Begriff der *Nachrichtenkomplexität*, also der Gesamtzahl verschickter Nachrichten, nicht aber mit der Anzahl Nachrichten auf dem kritischen Pfad oder anderen Maßen, die Aussagen über die parallele Ausführungszeit zuließen. Dies führt dazu, daß Verfahren, die einen (virtuellen) Ring als Kommunikationsstruktur verwenden, besonders gut abschneiden, obwohl sie eine Ausführungszeit in $\Omega(n)$ aufweisen. Da einige der in den fol-

genden Kapiteln vorgestellten Lastverteilungsverfahren Ausführungszeiten in $O(\log n)$ zulassen, würde die Übernahme dieser Verfahren einen Engpaß darstellen.

In DUTT UND MAHAPATRA (1993) wird ein Verfahren vorgestellt, das zumindest im besten Fall in Zeit $O(\log n)$ auf einem Hyperwürfel für parallele A*-Suche funktioniert. Allerdings wird für jede Nachricht eine Rückmeldung benötigt, die erst gegeben werden kann, wenn eine u.U. schon weit fortgeschrittene asynchrone Und-Reduktion zur Terminierungserkennung gestoppt wurde. In KUMAR ET AL. (1994) wird das *kreditbasierte Verfahren* (MATTERN, 1987, Abschnitt 4.9) für parallele Tiefensuche verwendet. Allerdings wird auch dort für jedes Stück abgegebene Arbeit eine Rückmeldung für die Terminierungserkennung benötigt. Diese darf erst abgegeben werden, wenn die damit assoziierte Arbeit abgearbeitet ist. Dadurch entstehen Kosten, die nicht durch gleichzeitige lokale Berechnungen überlappt werden können. Aus dieser Not machen FINKEL UND MANBER (1987) eine Tugend und nutzen diese Rückmeldungen für die Ergebnisberechnung. Außerdem läßt sich durch derart genaue Buchführung die Suche fehlertolerant gestalten, indem ausbleibende Teillösungen erneut berechnet werden. Bei vielen Anwendungen läßt sich die Ergebnisberechnung jedoch effizienter realisieren, so daß diese allein den Aufwand für das kreditbasierte Verfahren nicht rechtfertigt.

Abbildung 4.1 beschreibt durch einen Satz von Regeln, wie sich das im Mittel noch effizientere und sehr allgemeine *Doppelzählverfahren* (MATTERN, 1987, Abschnitt 4.6.1) so einfach und portabel implementieren läßt, daß das Argument der einfacheren Realisierung ringbasierter Verfahren entkräftet wird. Jedes PE zählt die lokal gesendeten und empfangenen Nachrichten. Die lokalen Zählerstände werden (asynchron) aufsummiert sobald alle PEs einmal arbeitslos geworden sind. Dies ist eine notwendige Terminierungsbedingung. Eine

When Initialization:
  var $c_{\text{out}} := 0, c_{\text{in}} := 0 : \mathbb{Z}$ (* local message counts *)
  var $s_{\text{out}} := 0, s_{\text{in}} := 0 : \mathbb{Z}$ (* global message counts *)
  var newCycleFlag := **true** : **boolean**

When Message Sent: $c_{\text{out}} := c_{\text{out}} + 1$

When Message Received: $c_{\text{in}} := c_{\text{in}} + 1$

When PE gets idle $\vee$ newCycleFlag changed to **true** :
  if newCycleFlag $\wedge$ PE is idle **then**
    supply $(c_{\text{out}}, c_{\text{in}})$ for an
    asynchronous computation of $\sum_{i<n} (c_{\text{out}}, c_{\text{in}}) \langle i \rangle$
    newCycleFlag := **false**

When computation of $\vec{s} := \sum_{i<n} (c_{\text{out}}, c_{\text{in}}) \langle i \rangle$ is completed:
  if $\vec{s} = (s_{\text{out}}, s_{\text{in}}) \wedge s_{\text{out}} = s_{\text{in}}$ **then**
    Termination detected
  else
    $(s_{\text{out}}, s_{\text{in}}) := \vec{s}$
    newCycleFlag := **true**

Abbildung 4.1: Terminierungerkennung mit dem Doppelzählverfahren.

hinreichende Terminierungsbedingung ist aber erst dann gegeben, wenn die Gesamtzahl gesendeter und empfangener Nachrichten gleich ist und auch ein nochmaliges Zählen das gleiche Ergebnis ergibt. Diese Formulierung hat den Vorteil, alle netzwerkabhängigen Aspekte des Algorithmus auf die Berechnung einer globalen Reduktion zu verlagern, einer wohlverstandenen Operation, die für andere Teile des Algorithmus oft ohnehin gebraucht wird und in vielen Kommunikationsbibliotheken auch bereits vorhanden ist.

Das Doppelzählverfahren kann nie mehr Aufwand verursachen als ein periodisches Zählen der arbeitslosen PEs. Im Mittel werden nur wenige Reduktionen durchgeführt, und die Latenzzeiten aller, außer den letzten beiden Reduktionen, können durch lokale Berechnungen verdeckt werden. Außerdem ist während dieser kritischen letzten Phase meist wenig anderweitiger Verkehr im Netzwerk, der die Latenz erhöhen könnte.

4.2 Angenäherte globale Summierung

Eine fundamentale Aufgabe beim Entwurf von Lastverteilungsalgorithmen besteht darin, einen Mittelweg zwischen zu ungenauem Lastausgleich und zu hohem Verteilungsaufwand zu finden. Eine wichtige Basistechnik zur Erreichung dieses Ziels ist es, den Lastverteiler adaptiv zu machen in dem Sinne, daß Kommunikation nur dann stattfindet, wenn tatsächlich Handlungsbedarf besteht. Dazu ist oft die Überwachung globaler Systemparameter wie Anzahl arbeitsloser Prozessoren, mittlere Last, o.ä. erforderlich.[1)] Alle diese Probleme lassen sich auf das möglichst genaue Verfolgen einer globalen Summe reduzieren: Gegeben Werte $x_i(t)$ $(i < n)$, bestimme eine möglichst genaue Näherung für $s(t) = \sum_{i<n} x_i(t)$. Wegen der endlichen Informationsausbreitungsgeschwindigkeit zwischen den PEs läßt sich dieses Problem nie perfekt lösen. Meist reicht jedoch eine ungefähre Bestimmung der Summe zu einem nicht allzuweit zurückliegenden Zeitpunkt.

Eine einfache Lösung besteht darin, die Summe periodisch durch eine globale Reduktionsoperation zu bestimmen. Leider führt dies selbst wieder zu signifikantem Kommunikationsaufwand, der in dem Sinne nicht adaptiv ist, daß Summen auch dann berechnet werden, wenn s sich kaum ändert. Eine mindestens ebenso einfache adaptive Lösung beruht darauf, daß ein PE dessen x-Wert sich ändert, eine entsprechende Nachricht an PE 0 schickt. Wenn der Wert sich auf vielen PEs ändert, führt dies allerdings zu einem Kommunikationsengpaß an PE 0.

Dieser Engpaß läßt sich durch Einsatz von Randomisierung in vielen Fällen vermeiden. Um zu einer präzisen und dennoch einfachen Beschreibung zu gelangen, wird die Problemstellung nun auf einen wichtigen Spezialfall eingeschränkt (eine Adaptierung für andere Varianten erscheint aber durchaus möglich.)

Problemstellung 4.1. Gegeben: Monoton wachsende Funktionen $x_i(t) \in \mathbb{N}^{\mathbb{R}_+}$, $x_i(0) = 0$ $(i < n)$, die die Entwicklung eines lokalen Wertes auf PE i beschreiben. Sei $s(t) := \sum_{i<n} x_i(t)$. Gesucht: Ein effizienter adaptiver Algorithmus, der das Überschreiten einer Schwelle $\overline{s}$ mit hoher Wahrscheinlichkeit schnell bemerkt, jedoch mit hoher Wahrscheinlichkeit nicht aktiv wird, solange $s(t) < (1-\varepsilon)\overline{s}$ für eine beliebiges $\varepsilon > 0$.

In LAUER (1995) wird das Problem dadurch angegangen, daß ein PE, dessen x_i sich erhöht, eine Nachricht an ein zufällig gewähltes PE schickt. Empfängt irgendein PE mehr als eine bestimmte Anzahl Nachrichten, so wird eine globale Reduktion gestartet. Dieses Verfahren hat eine sehr komplexe

[1)] Lastverteilung ist wohl nicht die einzige Anwendungsklasse, bei der dies nötig ist. Zum Beispiel ließen sich die unten beschriebenen Werkzeuge für die Realisierung von Leistungsanalysewerkzeugen mit hoher zeitlicher Auflösung und geringer Systembeeinflußung einsetzen.

Analyse, die ca. $2/3$ des Hauptteils der Arbeit ausmacht und läßt sich deshalb nur schwer an veränderte Gegebenheiten anpassen. Außerdem muß für jede Wertänderung eine globale Nachricht verschickt werden, und es funktioniert erst für recht große n zuverlässig.

Es geht aber auch deutlich einfacher: Ein PE, dessen x_i sich erhöht, schickt mit Wahrscheinlichkeit p eine Nachricht an PE 0 (für geeignetes p). Sei M die Anzahl bisher abgeschickter Nachrichten. Mit Hilfe elementarer Wahrscheinlichkeitstheorie, sowie der Sätze 3.5 und 3.7 über Summen von 0/1-Zufallsvariablen können unmittelbar folgende Aussagen festgestellt werden:

Lemma 4.2. $\mathbf{E}M = sp$

Lemma 4.3. $M \in \tilde{\mathrm{O}}(\max(sp, \log n))$

Lemma 4.4. $|sp - M| \in \tilde{\mathrm{O}}\left(\sqrt{sp \log n}\right)$

Der Quotient aus Anzahl Nachrichten und der Wahrscheinlichkeit des Absendens einer Nachricht, M/p, kann also als Schätzwert für den (unbekannten) wahren Wert von s benutzt werden; Nachrichtenstauungen halten sich in engen Grenzen, und der Schätzfehler wächst nur langsam mit sp. Diese Aussagen dienen nun als Basis für die Festlegung einer Strategie zur Lösung des oben gestellten Problems. Es wird ein geeignetes $p \in \Theta(\log n/\overline{s})$ festgelegt, und das Überschreiten der Schwelle wird dann angezeigt, wenn der Schätzwert für s die Mitte zwischen $(1-\varepsilon)\overline{s}$ und $\overline{s}$ überschreitet. Das Verfahren wird in einer einfachen Version geschildert, die den Schönheitsfehler hat, daß es mit einer Wahrscheinlichkeit $\geq n^{-\beta}$ fehlschlägt, wobei β zwar beliebig groß gemacht werden kann, aber von im Programm einzustellenden Parametern abhängt. In konkreten Anwendungen läßt sich dieses Problem aber meist lösen.

Satz 4.5. *Sei $\beta > 0$ beliebig. Dann gibt es ein c, so daß für $p = c\frac{\log n}{\overline{s}}$, $\overline{m} := (1-\frac{\varepsilon}{2})\overline{s}p$ und hinreichend große n gilt,*

$$s \geq \overline{s} \Rightarrow \mathbf{P}[M < \overline{m}] \leq n^{-\beta} \qquad \textit{und} \qquad s < (1-\varepsilon)\overline{s} \Rightarrow \mathbf{P}[M \geq \overline{m}] \leq n^{-\beta} \ .$$

Beweis.
Fall $s \geq \overline{s}$: Sei O.B.d.A. $s = \overline{s}$. (Zusätzliche Versuche können M nur erhöhen.) Dann gilt

$$\begin{aligned}\mathbf{P}[M < \overline{m}] =& \mathbf{P}[sp - M > sp - \overline{m}] = \mathbf{P}\left[sp - M > c\frac{\varepsilon}{2}\log n\right] \\ \leq& \mathbf{P}\left[|sp - M| > c\frac{\varepsilon}{2}\log n\right] = \mathbf{P}\left[|sp - M| > \sqrt{c}\frac{\varepsilon}{2}\sqrt{sp\log n}\right] \ .\end{aligned}$$

Bei Wahl eines hinreichend großen c folgt aus Lemma 4.4 für hinreichend große n

$$\mathbf{P}[M < \overline{m}] \leq n^{-\beta} \ .$$

Fall $s < (1-\varepsilon)\overline{s}$: Analog zum Fall $s \geq \overline{s}$. Nun wird mit dem Ansatz $s = (1-\varepsilon)\overline{s}$ und $\mathbf{P}[M \geq \overline{m}] = \mathbf{P}[M - sp \geq \overline{m} - sp]$ gearbeitet. □

Weiterhin folgt aus Lemma 4.3, daß für $p \in \mathrm{O}(\log n/\overline{s})$ und $s \in \mathrm{O}(\overline{s})$ die Anzahl empfangener Nachrichten M in $\tilde{\mathrm{O}}(\log n)$ liegt. Die Zeit die verstreicht, bis hinreichend viele Nachrichten angekommen und verarbeitet sind liegt in $\mathrm{O}(d+M) \in \tilde{\mathrm{O}}(d+\log n)$ falls keine aus anderen Quellen stammenden Nachrichten das Kommunikationsnetz verstopfen.

4.2.1 Anwendungen und Erweiterungen

In dieser Arbeit wird das oben beschriebene Verfahren in Abschnitt 7.4 eingesetzt, um eine (monoton fallende) Prozessorauslastung abzuschätzen. Bei Überschreiten einer maximalen „Arbeitslosenquote“ wird eine Lastverteilungsphase ausgelöst. Für diesen Fall läßt sich das Verfahren noch robuster machen. Würfeln arbeitslose PEs periodisch, ob sie eine Zählnachricht senden sollen, so gibt es einen Zeitpunkt in $\tilde{\mathrm{O}}(d+\log n)$ zu dem das Überschreiten der Schwelle festgestellt wird. Ein „falscher Alarm“ wird dadurch zwar möglich, aber dieser wird nur auftreten, wenn die PE-Auslastung über einen langen Zeitraum hoch ist. Man kann zeigen, daß in diesem Fall das gelegentliche vorzeitige Auslösen einer Lastverteilungsphase sogar vorteilhaft ist.

In SANDERS (1995c) wird ein Algorithmus zum engpaßfreien parallelen Zugriff auf eine Priority queue[2)] beschrieben. Hier kann die angenäherte Summierung eingesetzt werden, um aus einem synchron formulierten Algorithmus einen (quasi-)kontinuierlich ablaufenden Algorithmus zu machen. Es geht dabei darum festzustellen, wann die Anzahl aufgelaufener „Entferne-Minimum“-Anforderungen groß genug ist, um einen (kollektiven) Zugriff auf die verteilte Datenstruktur zu rechtfertigen.

Ein sehr ähnliches Problem tritt auf, wenn Lastverteilungsalgorithmen für parallele Tiefensuche, die für SIMD-Architekturen entworfen wurden, für MIMD-Rechner adaptiert werden sollen (Siehe auch Abschnitt 6.1). Wann sich eine (wieder kollektive) Lastverteilungsphase lohnt, hängt von der PE-Auslastung, aber auch von deren Entwicklung seit der letzten Phase und den (u.U. variablen) geschätzten Kosten für die nächste Lastverteilungsphase ab (POWLEY ET AL., 1993; KARYPIS UND KUMAR, 1994). Die zu beobachtende Auslastungsschwelle ist also nicht konstant. Dies kann aber ohne weiteres durch entsprechende Änderung der Schwelle $\overline{m}$ für die Anzahl empfangener Nachrichten berücksichtigt werden.

Eine gewisse Schwierigkeit ergibt sich, wenn $s(t)$ nicht monoton ist, da nicht bekannt ist, in welche Richtung der Schätzwert für s durch die im Netz befindlichen Nachrichten verändert wird. Der in LAUER (1995) beschriebene Lastverteilungsalgorithmus benötigt z.B. eine Abschätzung der mittleren Last. Diese kann sich erhöhen und senken. Das hier beschriebene Verfahren kann aber verwendet werden, um den *Betrag* aller Laständerungen abzuschätzen. Sobald dieses Maß eine signifikante Änderung der Durchschnittslast *möglich* erscheinen läßt, kann durch eine echte Summierung der Gesamtlast Gewißheit erlangt werden. Verändert sich die Summe s aber hinreichend langsam (wie auch in LAUER (1995)), so kann eine Verminderung von s einfach negativ gezählt werden. Auf eine genaue Durchrechnung dieser Algorithmenvariante soll hier verzichtet werden. Jedoch steht zu vermuten, daß die Anzahl notwendiger Summierungsoperationen dadurch stark reduziert werden kann. Ganz auf Summierung verzichtet werden kann aber wohl nicht, da die Beobachtungsfehler sich wie bei einem „Random walk“ langsam aufsummieren.

Erhöht sich der Wert von s um mehr als 1 (z.B. k), so kann einfach k mal unabhängig voneinander entschieden werden, ob eine Nachricht gesendet wird. Tritt dies häufig auf, kann der gleiche Effekt effizienter erreicht werden, indem nicht nur eine einfache Nachricht sondern ein $b(k,p)$ binomialverteilter Zufallswert an PE 0 geschickt und dort als Multiplizität der Nachricht interpretiert wird. Einigermaßen effiziente Verfahren zur Erzeugung binomialverteilter Zufallszahlen finden sich z.B. in PRESS ET AL. (1992).

Die Genauigkeit der Beobachtung der Summe s kann auf Netzen mit relativ großem Durchmesser verbessert werden, indem die Nachrichtensendewahrscheinlichkeit p größer gewählt wird. Zum Beispiel wäre bei einem r-dimensionalem Gitter jedes p in $\mathrm{O}\left(n^{1/r}/\overline{s}\right)$ möglich, ohne daß sich daraus ein Kommunikationsengpaß ergäbe. Noch genauere Ergebnisse lassen sich erzielen, indem die Nachrichtensendewahrscheinlichkeit stark vergrößert wird und die dadurch gestiegene Zahl Nachrichten auf mehrere PE verteilt werden. Diese können dann z.B. periodische Summierungen durchführen oder in

[2)] Also eine Datenstruktur, die mit einer Priorität versehene Werte speichert und die Operationen „Einfügen“ und „Entferne Minimum“ unterstützt.

Abschnitt 2.4.2 erwähnten verteilten globalen Zähler verwenden. Die Beobachtungsgenauigkeit läßt sich so auf Kosten eines gestiegenen Nachrichtenaufkommens beliebig erhöhen.

4.3 Zufallsgeneratoren

Schon im sequentiellen Fall sind Zufallsgeneratoren eine heikle und nichttriviale Angelegenheit. (Siehe z.B. KNUTH (1981); PRESS ET AL. (1992).) Es gibt jedoch sequentielle Pseudozufallsgeneratoren, die sich für viele Anwendungen bewährt haben. Außerdem können die meisten von sequentiellen Generatoren erzeugten Zahlenfolgen effizient in disjunkte Teilfolgen zerlegt werden (L'ECUYER, 1990; HENNECKE, 1994), die unabhängig voneinander parallel erzeugt werden können. Allerdings ist bei der Benutzung dieser Verfahren noch mehr Vorsicht geboten als schon im sequentiellen Fall (MATTEIS UND PAGNUTTI, 1990). Zum Beispiel reicht die Periodenlänge vieler verbreiteter sequentieller Zufallsgeneratoren für parallele Anwendungen nicht aus.

Auf der nächst höheren Abstraktionsstufe der Generierung komplexerer Zufallsobjekte gibt es allerdings viele wenig untersuchte Fragestellungen. In dieser Arbeit werden zum Beispiel verschiedene Typen von Zufallspermutationen benötigt auf deren Berechnung in den Abschnitten 4.3.1 und 4.3.2 eingegangen wird.

4.3.1 Zufallspermutationen über $\mathbb{N}_n$

In Kapitel 7 müssen Teilprobleme zwischen den PEs zufällig ausgetauscht werden, um Lastagglomerationen aufzulösen:

Problemstellung 4.6 (Zufallspermutationen). Gesucht: Ein zufälliges Element π aus der Permutationsgruppe S_n (das heißt, einer bijektiven Abbildung von $\mathbb{N}_n$ auf sich selbst.) Jedes Element soll mit der gleichen Wahrscheinlichkeit $\frac{1}{n!}$ ausgewählt werden. Der Wert $\pi(i)$ soll auf PE i verfügbar sein.

Im Sequentiellen lassen sich solche Permutationen recht einfach in Zeit $\Theta(n)$ generieren (KNUTH, 1981). Der dort beschriebene Algorithmus ist allerdings inhärent sequentiell und für eine parallele Implementierung ungeeignet. Weiterhin gibt es ausgeklügelte Algorithmen für CRCW-PRAMS[3)] (HAGERUP, 1991), QRQW-PRAMS[4)] (GIBBONS ET AL., 1994) und CREW-PRAMS [5)] (CZUMAJ ET AL., 1996) aber diese erscheinen für nachrichtengekoppelte Rechner nur bedingt geeignet. Die (asymptotisch) besten bisher bekannten Algorithmen für diese Architektur beruhen auf Sortieren.

Abbildung 4.2 gibt einen einfachen parallelen Algorithmus zur Erzeugung einer Zufallspermutation an. Zunächst schickt jedes PE seine Nummer an ein zufällig gewähltes PE. Nachdem die auf einem PE zusammengekommenen Werte einer lokalen sequentiellen Zufallspermutation unterworfen wurden, werden die Elemente mit Hilfe einer Präfixsummenberechnung global durchnumeriert. Die Werte werden dann an das PE mit der so ermittelten Nummer geschickt. Auf diese Weise erhält jedes PE einen anderen Wert aus $\mathbb{N}_n$ der als $\pi(i_{\mathrm{PE}})$ interpretiert wird.

[3)] Concurrent Read Concurrent Write Parallel Random Access Machine

[4)] Queue Read Queue Write Parallel Random Access Machine

[5)] Concurrent Read Exclusive Write Parallel Random Access Machine

(* Determine a random permutation π and return $\pi(i_{\text{PE}})$. *)
Function randPerm : $\mathbb{N}_n$
  send i_{PE} to a random PE (* uniformly distributed *)
  store incoming values in $e_0, \ldots, e_{k-1}$
  permute the e_j randomly (* sequentially *)
  $\Delta := \sum_{i<i_{\text{PE}}} k\langle i\rangle$ (* prefix sum *)
  send e_i to PE $\Delta + i$ (for $i \in \mathbb{N}_k$)
  receive exactly one value and return it

Abbildung 4.2: Erzeugung einer Zufallspermutation

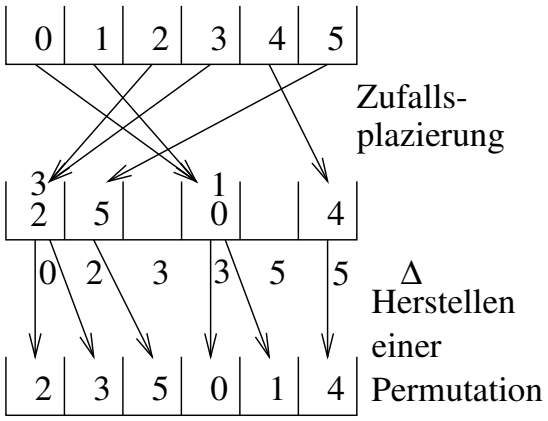

Abbildung 4.3: Beispiel für Zufallspermutationserzeugung.

Abbildung 4.3 stellt den Ablauf des Algorithmus an einem Beispiel dar. Es stellt sich die Frage, ob dieser Algorithmus tatsächlich *gleichverteilte* Zufallspermutationen erzeugt und ob er dies auch effizient tut. Die folgenden zwei Sätze beantworten diese Frage:

Satz 4.7. *Jede mögliche Permutation über n Eingabeelementen wird mit gleicher Wahrscheinlichkeit* $1/n!$ *von Algorithmus 4.2 erzeugt.*

Beweis. Sei π' eine beliebige aber feste Permutation. Es werden nun die möglichen Programmabläufe von Algorithmus 4.2 betrachtet. Sei π die erzeugte Permutation. Nach einer grundlegenden Formel über bedingte Wahrscheinlichkeiten (z.B. HINDERER (1988) Seite 9.4) gilt

$$\mathbf{P}\left[\pi = \pi'\right] = \mathbf{P}\left[\bigwedge_{i<n} \pi(i) = \pi'(i)\right] = \prod_{i<n} \mathbf{P}\left[\pi(i) = \pi'(i) \mid \bigwedge_{j<i} \pi(j) = \pi'(j)\right] \quad .$$

Da $\frac{1}{n!}$ als $\prod_{i<n} \frac{1}{n-i}$ geschrieben werden kann, genügt es zu zeigen, daß

$$\mathbf{P}\left[\pi(i) = \pi'(i) \mid \bigwedge_{j<i} \pi(j) = \pi'(j)\right] = \frac{1}{n-i} \quad .$$

Sei $K := (k\langle 0\rangle, \ldots, k\langle n-1\rangle)$ der Vektor der Multiplizitäten, die in Algorithmus 4.2 berechnet werden. Es wird nun eine Fallunterscheidung nach den möglichen Werten von K gemacht und gezeigt, daß für jedes mögliche K' gilt

$$P_i := \mathbf{P}\left[\pi(i) = \pi'(i) \mid K = K' \wedge \bigwedge_{j<i} \pi(j) = \pi'(j)\right] = \frac{1}{n-i} \ .$$

Sei l so gewählt, daß $\sum_{j<l} k\langle j\rangle < i \leq \sum_{j\leq l} k\langle j\rangle$. Offenbar gilt $\pi(i) = \pi'(i)$ genau dann, wenn der Wert $\pi(i)$ zunächst an PE l gesendet wird und dann lokal an die Stelle $i - \Delta\langle l\rangle$ permutiert wird. Es sind unter den gegebenen Bedingungen noch $n-i$ Werte zu plazieren, wobei wegen $K = K'$, $k' := k + \Delta\langle l\rangle - (n-i)$ davon an PE l zu schicken und dort noch lokal zu permutieren sind. Folglich wird $\pi(i)$ mit Wahrscheinlichkeit $\frac{k'}{n-i}$ an PE l gesendet und in diesem Fall mit Wahrscheinlichkeit $\frac{1}{k'}$ an die Stelle $i - \Delta\langle l\rangle$ permutiert. P_i ergibt sich aus dem Produkt dieser beiden Wahrscheinlichkeiten zu $\frac{1}{n-i}$. □

Satz 4.8. *Die Laufzeit von Algorithmus 4.2 ist in* $\tilde{\mathrm{O}}(T_{\mathrm{rout}} + T_{\mathrm{coll}})$.

Beweis. Das Verschicken von Werten an zufällig gewählte PEs kostet nach Satz 2.5 Zeit T_{rout}. Nach Satz 3.14 über Zuordnungsprobleme ist die maximale Zahl von Werten, die auf einem PE landen in $\tilde{\mathrm{O}}\left(\frac{\log n}{\log\log n}\right)$ und folglich ist die lokale Permutation in Zeit $\tilde{\mathrm{O}}\left(\frac{\log n}{\log\log n}\right) \subseteq \tilde{\mathrm{O}}(T_{\mathrm{coll}})$ erledigt.[6] Δ kann durch eine Präfixsummenberechnung mit Zeitaufwand T_{coll} bestimmt werden. Der letzte Schritt des Algorithmus ist ein Datentransportproblem, das mit dem in LEIGHTON (1992) beschriebenen *Spreading* verwandt ist. Dies erscheint intuitiv eher leichter als z.B. das in Satz 2.5 beschriebene Empfangen von einer zufälligen Quelle. Deshalb sei hier nur kurz angedeutet, wie sich zeigen läßt, daß dieses Datentransportproblem in Zeit $\tilde{\mathrm{O}}(T_{\mathrm{rout}} + \log n)$ lösbar ist: Das PE-Netzwerk wird in zusammenhängende Teilnetzwerke der Größe $\mathrm{O}(\log n)$ partitioniert. Nach Satz 3.14 liegen auf keinem Teilnetzwerk mehr als $\tilde{\mathrm{O}}(\log n)$ Werte. Diese können durch Transport innerhalb des Teilnetzwerkes in Zeit $\tilde{\mathrm{O}}(\log n)$ so umverteilt werden, daß kein PE mehr als $\tilde{\mathrm{O}}(1)$ Werte behält. Anschließend kann das verbleibende globale Datentransportproblem nach Satz 2.5 in Zeit $\tilde{\mathrm{O}}(T_{\mathrm{rout}})$ gelöst werden. □

Die Effektivität dieses Algorithmus läßt sich auch praktisch nachprüfen. In GIBBONS ET AL. (1994) sind drei PRAM-Algorithmen für die MasPar MP-1 adaptiert. Für $n = 2^{14}$ benötigt ein auf (Integer?)-Sorting basierendes Verfahren 11.25 ms. Zwei Implementierungen, die mit mehrfachem zufälligem Zuordnen arbeiten, laufen in 8.02 ms bzw. 7.57 ms ab. Der hier beschriebene Algorithmus mit einfachem zufälligem Zuordnen und einer Präfixsumme benötigt nur 5.16 ms.

4.3.2 Pseudozufallspermutationen

Im vorangegangenen Abschnitt wird beschrieben, wie aus parallel erzeugten Pseudozufallsfolgen Permutationen konstruiert werden können. Dieses Verfahren ist etwas unbefriedigend,

[6] Dieser Wert bietet auch Spielraum für eine Analyse in einem Modell bei dem nur ein einziges Zufallsbit in konstanter Zeit berechnet werden kann. Für eine Permutation von k Werten werden $\Theta(k \log k)$ Zufallsbits benötigt und folglich ergäbe sich dann ein Zeitaufwand in $\tilde{\mathrm{O}}\left((\frac{\log n}{\log\log n})(\log \frac{\log n}{\log\log n})\right) = \tilde{\mathrm{O}}(\log n)$ für die lokale Permutation.

weil die $\Theta(n \log n)$ Zufallsbits, die benötigt werden, um eine echte Zufallspermutation zu erzeugen, nur scheinbar zur Verfügung stehen und für große n in Wirklichkeit aus Pseudozufallsfolgen mit so kurzer Periodenlänge erzeugt werden müssen, daß nur ein Bruchteil aller möglichen Permutationen erzeugt werden kann. Wenn eine Gleichverteilung auf den Permutationen also ohnehin nur grob angenähert werden kann, liegt es nahe, Pseudozufallspermutationen direkt und ohne großen Koordinationsaufwand zu erzeugen. Leider gibt es für diesen Ansatz wenig bekannte Ergebnisse, so daß Einfallsreichtum und vor allem Vorsicht gefragt sind.

Ein besonders günstiger Fall liegt vor, wenn eine einfach zu handhabende Teilmenge von Permutationen identifiziert werden kann, die garantiert, daß der Algorithmus sich so verhält wie für echte Zufallspermutationen. So zeigt sich in Abschnitt 6.1.3, daß dort schon eine zyklische Verschiebung um eine zufällig gewählte Distanz genügt.

In Kapitel 8 wird eine einzige pseudozufällige Permutation über $\mathbb{N}_{2^{h'}}$ ($2^{h'} \gg n$) benötigt. Hier bietet sich z.B. ein linearer Kongruenzengenerator der Form $x_{k+1} = ax_k + c \bmod 2^{h'}$ an. Laut KNUTH (1981) ergeben sich daraus bei geeigneter Wahl von a und c brauchbare Pseudozufallsfolgen mit Periodenlänge $2^{h'}$. Wird $\pi(k) := x_k$ definiert, so entsteht also eine Permutation. Wie „zufällig“ diese Permutation ist, ist allerdings eine offene Frage, denn die „Brauchbarkeit“ als Pseudozufallsfolge impliziert noch nicht, daß die Benutzung der *gesamten* Folge als Permutation eine gute Idee ist.

Eine andere Möglichkeit besteht darin, ein primitives Polynom p modulo zwei vom Grad h' zu verwenden und auszunutzen, daß für jedes l, das teilerfremd zu $2^{h'} - 1$ ist, gilt, daß $x^l \bmod p$ ein erzeugendes Element der multiplikativen Gruppe von $GF(2^{h'})$ ist. Das heißt, die Folge $y_k := ((x^l)^1, \ldots, (x^l)^{2^{h'}-1}) \bmod p$ zählt die Polynome modulo p (bis auf das Nullpolynom) auf. Durch Einfügen der 0 an eine beliebige Stelle in der Folge ergibt sich eine Permutation π, indem die Koeffizienten von y_k als Ziffern einer Binärzahl interpretiert werden, die $\pi(k)$ codiert. Die notwendigen Operationen können effizient über Bitoperationen implementiert werden. Das Verfahren hat den Vorteil, daß sich durch Variation von l die erzeugte Permutation variieren läßt und daß in kryptographischem Sinne als sicher angesehene Zufallsfolgen erzeugt werden.[7)] Ebenfalls auf Polynomen modulo zwei beruhen *Sobol-Folgen* wie sie zur Beschleunigung von Monte-Carlo Integrationsverfahren verwendet werden (PRESS ET AL., 1992). Diese lassen sich sehr effizient berechnen und können ebenfalls zu Permutationen über $\mathbb{N}_{2^m}$ umfunktioniert werden.

4.4 Zusammenfassung

Die effiziente Beobachtung globaler Systemzustände ist ein umfangreiches und interessantes Gebiet. Die hier beschriebenen Algorithmen zur Terminierungserkennung und zur angenäherten globalen Summierung sind nur ein kleiner Ausschnitt aus einem weiten Spektrum von verwandten Problemstellungen.

Zufallszahlengeneratoren werden selbst im sequentiellen Fall zu oft als selbstverständlich hingenommen. Für parallele Algorithmen gibt es ein Anzahl weiterer Dinge, die zu beachten sind. Außerdem versteckt sich hinter einfach aussehenden Problemen, wie dem Erzeugen von Zufallspermutationen, oft ein unerwarteter Koordinationsaufwand. In diesem Bereich

[7)] Für diesen Hinweis möchte ich Torsten Minkwitz und Jörn Müller-Quade danken.

erscheint weitere Forschung angebracht, um kombinatorische Pseudozufallsstrukturen direkt – ohne den Umweg über unabhängige Pseudozufallsfolgen – zu erzeugen.

Ergebnisse

Das in Abschnitt 4.2 entwickelte Verfahren zur Beobachtung von Summen ist das erste Verfahren, das nur bei signifikanten Wertänderungen Kommunikationsaufwand verursacht und selbst dann nur wenige Nachrichten verschickt.

Ein so einfacher und effizienter Algorithmus zum Erzeugen von Zufallspermutationen durch Datentransport und Präfixsummen ist vorher noch nicht beschrieben worden. (Allerdings ist in REIF (1985) ein EREW-PRAM[8)]-Algorithmus beschrieben, der auf „Integer sorting“ beruht und ähnlich funktioniert.[9)]) Außerdem fehlt in den meisten Veröffentlichungen (oft wohl aus Platzgründen) der Nachweis, daß eine Gleichverteilung auf den Permutationen erzeugt wird.[10)]

Das Doppelzählverfahren zur Terminierungserkennung ist zwar nicht neu, doch seine Vorteile für baumförmige Berechnungen sind bisher nicht erkannt worden. Die Formulierung durch asynchrone kollektive Operationen ist hervorzuheben, weil sie ebenso einfach aber mindestens so portabel und deutlich effizienter als die in der Literatur dominierende ringbasierte Variante ist.

[8)] Exclusive Read Exclusive Write Parallel Random Access Machine

[9)] Für diesen Hinweis möchte ich Artur Czumaj von der Universität Paderborn danken.

[10)] Daß dies nicht völlig trivial ist, mag man daran sehen, daß in der Newsgroup `comp.theory` alle paar Monate nach sequentiellen Algorithmen für Zufallspermutationen gefragt wird und die ersten Antworten regelmäßig falsche Algorithmen enthalten.

Kapitel 5

Untere Schranken

Hauptziel dieser Arbeit ist es, gute Lastverteilungsalgorithmen für baumförmige Berechnungen zu finden. Dafür muß aber klarer gemacht werden, was ein „guter“ Algorithmus ist. Ein objektiver Ansatz besteht darin, zunächst untere Schranken herzuleiten, die kein Algorithmus oder kein Algorithmus in einer bestimmten Klasse übertreffen kann. Ein Algorithmus kann dann daran gemessen werden, wie nah er den unteren Schranken kommt. In Abschnitt 5.1 werden zunächst einige einfach zu begründende Schranken hergeleitet, die aber schon interessante Einsichten erlauben. Einige dieser Schranken gelten sogar für *alle* baumförmigen Berechnungen, können also auch bei „gutmütigen“ Probleminstanzen nicht unterschritten werden. Dann zeigt Abschnitt 5.2, daß Algorithmen, die auf empfängerveranlaßtem Spalten beruhen, u.U. eine große Zahl von Teilproblemen übertragen müssen. Deshalb ist die Netzwerkbandbreite ein entscheidender Faktor. In Abschnitt 5.3 stellt sich heraus, daß Algorithmen, die das Wurzelproblem zerlegen, indem sie es mehrfach bis zu einer fest vorgegebenen Tiefe aufspalten, sehr viele Teilprobleme verarbeiten müssen. Dadurch lassen sich die statischen Lastverteilungsalgorithmen aus Kapitel 8 und eine Reihe in der Literatur betrachteter dynamische Lastverteilungsalgorithmen besser verstehen. In Abschnitt 5.4 werden diese Ergebnisse ausgewertet und kurz mit existierenden Algorithmen verglichen.

Eine grundlegende Technik für den Nachweis von unteren Schranken besteht darin, baumförmige Berechnungen anzugeben, die kein Lastverteilungsalgorithmus schneller als eine bestimmte Schranke bearbeiten kann. Es wird eine besonders einfach aufgebaute Klasse von baumförmige Berechnungen verwendet, die durch Binärbäume repräsentiert werden, deren Blätter atomare Teilprobleme darstellen und deren innere Knoten alle Grad zwei haben. Wird die Spaltoperation „split“ auf einen nichtatomaren Baum angewandt, so werden die beiden Teilbäume zurückgeliefert, die am Wurzelknoten hängen. Im Falle eines Blattes wird ein leeres plus ein atomares Teilproblem produziert. Die sequentielle Bearbeitung duch „work“ traversiert den Baum durch Tiefensuche von links nach rechts und markiert die abgearbeiteten Teile als fertig bearbeitet. Dies hat aber keinen Einfluß auf die Arbeitsweise der Spaltoperation. Der gesamte sequentielle Berechnungsaufwand ist in den Blättern konzentriert.

5.1 Einfache Schranken

Begonnen sei mit einigen einfachen Beobachtungen, die für alle baumförmigen Berechnungen gelten, wenn die Parameter l, T_{atomic} und T_{split} nicht nur obere Schranken sind, sondern asymptotisch genaue Angaben darstellen.

Satz 5.1. *Für alle baumförmigen Berechnungen gilt* $T_{\text{par}} \in \Omega\left(\frac{T_{\text{seq}}}{n} + T_{\text{atomic}}\right)$.

Beweis. Da durch das Aufspalten von Berechnungen die sequentielle Arbeit nicht weniger wird, muß diese irgendwo erledigt werden. Ferner kann es Teilprobleme der Größe T_{atomic} geben, die nicht weiter unterteilbar sind. Diese müssen irgendwo sequentiell bearbeitet werden. □

Diese Beobachtung war der Grund für die Annahme in Abschnitt 2.3.1, daß $T_{\text{seq}} \in \Omega(nT_{\text{atomic}})$. Sonst ist nämlich keine kosteneffiziente parallele Berechnung möglich. Um eine konstante Effizienz zu erreichen, muß die Berechnung außerdem irgendwie unter $\Omega(n)$ PEs aufgeteilt werden. Der dafür nötige Aufwand ermöglicht eine Verschärfung der Schranke von Satz 5.1.

Satz 5.2. *Für alle baumförmigen Berechnungen gilt: Soll eine Effizienz in* $\Omega(1)$ *erreicht werden, so ist*

$$T_{\text{par}} \in \Omega\left(\frac{T_{\text{seq}}}{n} + T_{\text{atomic}} + d + l + T_{\text{split}} \log n\right) \quad .$$

Beweis. Auf allen gebräuchlichen Verbindungsnetzwerken müssen Nachrichten über eine Entfernung von $\Omega(d)$ verschickt werden, um $\Omega(n)$ PEs zu erreichen. Eine dieser Nachrichten muß Länge l haben. Schließlich sind selbst bei einer perfekt funktionierenden Spaltfunktion $\Omega(\log n)$ aufeinanderfolgende Aufspaltungen nötig um eine Teilproblemgröße in $O\left(\frac{T_{\text{seq}}}{n}\right)$ zu erzwingen.

□

Es gibt Probleminstanzen, für die weit mehr als $\Omega(\log n)$ Aufspaltungen nötig sind, um hinreichend kleine Teilprobleme zu erzwingen. Für diese gilt die folgende weiter verschärfte Schranke:

Satz 5.3. *Es gibt baumförmige Berechnungen für die gilt: Soll eine Effizienz in* $\Omega(1)$ *erreicht werden, so ist*

$$T_{\text{par}} \in \Omega\left(\frac{T_{\text{seq}}}{n} + T_{\text{atomic}} + d + l + \left(h - \log \frac{T_{\text{seq}}}{nT_{\text{atomic}}}\right) T_{\text{split}}\right) \quad .$$

Beweis. Abbildung 5.1 zeigt ein Beispiel für eine Probleminstanz, bei der $h - \log \frac{T_{\text{seq}}}{T_{\text{atomic}}}$ aufeinanderfolgende Spaltoperationen zu keiner Problemaufteilung führen. Danach sind dann noch $\Omega(\log n)$ Aufspaltungen nötig, um ausreichend Parallelität für $\Omega(n)$ PEs zu produzieren. □

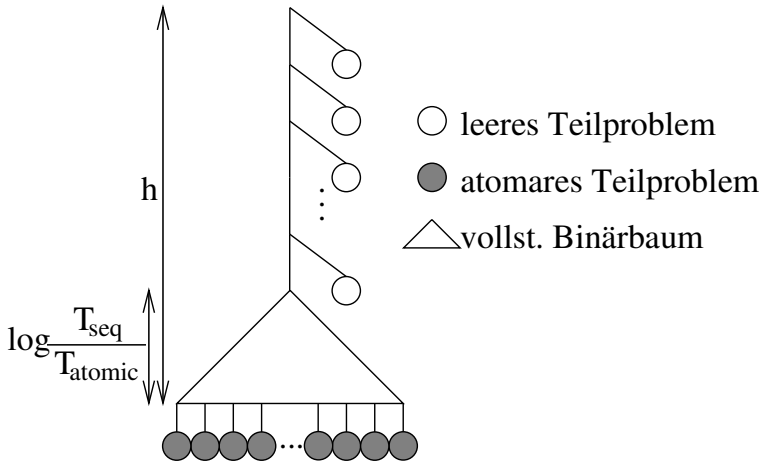

Abbildung 5.1: Baumförmige Berechnung mit vielen Aufspaltungen auf dem kritischen Pfad.

Die Aussagen in diesem Abschnitt geben bereits einen wichtigen Hinweis darauf, welche Aspekte bei der Parallelisierung entscheidend sein können. Bei sehr grobgranularen Berechnungen, kann das Verhältnis von T_{seq} zu T_{atomic} den nutzbaren Parallelismus beschränken. In anderen Fällen können die Netzwerklatenz oder die Länge einer Teilproblembeschreibung die gleiche Rolle spielen. In vielen Fällen wird aber der Aufwand für h-maliges Aufspalten eines Teilproblems die anderen Einflußfaktoren dominieren. In den folgenden Abschnitten wird sich die Tendenz noch verstärken, daß die mit der Problemaufspaltung verknüpften Kosten eine kritische Größe darstellen.

5.2 Kommunikationsumfang bei empfängerveranlaßtem Spalten

Zur Vereinfachung der Darstellung sei für diesen Abschnitt angenommen, daß n eine Zweierpotenz ist. Außerdem sei festgelegt, daß eine Effizienz von mindestens $\frac{1}{2}$ angestrebt wird Eine Anpassung für allgemeine n und beliebige vorgegebene Effizienzen ist nicht schwierig.

Satz 5.4. *Es gibt baumförmige Berechnungen, für die jedes Lastverteilungsverfahren mindestens* $\frac{n}{2}\left(h - \log \frac{T_{\text{seq}}}{T_{\text{atomic}}}\right)$ *Problemaufspaltungen vornehmen muß, um eine Effizienz* $\geq \frac{1}{2}$ *zu erreichen.*

Beweis. Abbildung 5.2 zeigt eine solche Probleminstanz. Es gibt tief unten im Baum $\frac{n}{2}$ Teilbäume, die jeweils eine sequentielle Arbeit von $2\frac{T_{\text{seq}}}{n}$ repräsentieren. Damit eine Effizienz von $\frac{1}{2}$ überschritten wird, müssen *alle* diese Teilbäume mindestens einmal aufgespalten werden. Bevor dies geschieht, müssen aber die $\frac{n}{2} - 1$ Aufspaltungen auf den obersten $\log n - 1$ Ebenen des Spaltbaums und die $\frac{n}{2}\left(h - (\log n - 1) - \log \frac{2T_{\text{seq}}}{nT_{\text{atomic}}}\right)$ Abspaltungen von leeren Teilproble-

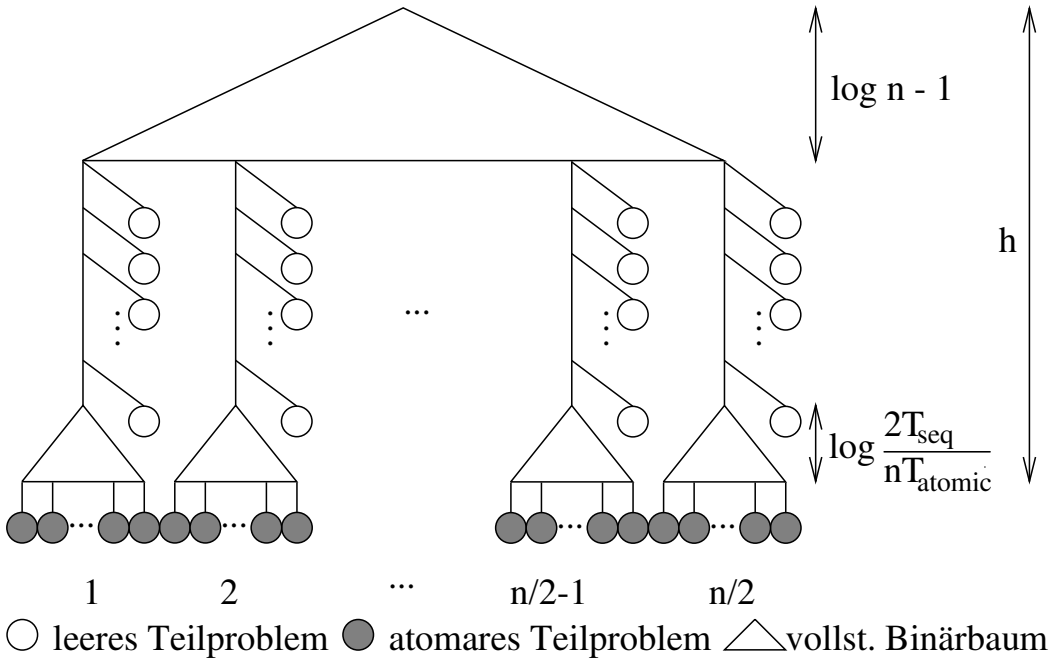

Abbildung 5.2: Probleminstanz mit sehr vielen nötigen Aufspaltungen.

men vorgenommen werden. Insgesamt also

$$\begin{aligned}\frac{n}{2}+\left(\frac{n}{2}-1\right)+\frac{n}{2}\left(h-(\log n-1)-\log\frac{2T_{\text{seq}}}{nT_{\text{atomic}}}\right)&\\ =\frac{n}{2}\left(2+h-\log n+1-1+\log n-\log\frac{T_{\text{seq}}}{T_{\text{atomic}}}\right)-1&\\ \geq\frac{n}{2}\left(h-\log\frac{T_{\text{seq}}}{T_{\text{atomic}}}\right)\quad.&\end{aligned}$$

□

Beruht das Lastverteilungsverfahren auf empfängerveranlaßtem Spalten, so ist jede Lastaufspaltung außerdem an eine Lastübertragung gebunden, und es ergibt sich die folgende Aussage:

Korollar 5.5. *Es gibt baumförmige Berechnungen, für die jedes auf empfängerveranlaßtem Spalten beruhende Lastverteilungsverfahren mindestens* $\frac{n}{2}(h-\log\frac{T_{\text{seq}}}{T_{\text{atomic}}})$ *erfolgreiche Lastanfragen abwickeln muß, um ein Effizienz* $\geq\frac{1}{2}$ *zu erreichen.*

Leider ist damit nichts darüber ausgesagt, ob die Lastübertragungen lokal oder global zu sein haben. Immerhin kann die Laufzeitschranke aus Satz 5.3 etwas verschärft werden. Vor allem für mehrstufige Verbindungsnetzwerke und Busse, bei denen es „lokale“ Kommunikation nicht gibt, ergeben sich interessante Aussagen.

Korollar 5.6. *Soll eine Effizienz in* $\Omega(1)$ *erreicht werden, so gilt*

$$T_{\text{par}}\in\Omega\left(\frac{T_{\text{seq}}}{n}+T_{\text{atomic}}+d+\left(h+\log n-\log\frac{T_{\text{seq}}}{T_{\text{atomic}}}\right)(T_{\text{split}}+l)\right)\quad.$$

Für mehrstufige Verbindungsnetzwerke (ohne Fat trees) steht $(T_{\text{split}} + l + \log n)$ *statt* $(T_{\text{split}} + l)$ *und für Busse sogar* $(T_{\text{split}} + nl)$.

5.3 Anzahl Teilprobleme bei „blindem“ Spalten

Im Mittelpunkt dieses Abschnitts steht wieder die Beobachtung, daß erst Teilprobleme mit $\text{gen}(P)$ nahe bei h garantiert hinreichend klein sind, um eine Bearbeitung auf einem einzigen PE zu rechtfertigen:

Satz 5.7. *Jeder Lastverteilungsalgorithmus, der das Wurzelproblem bis zu einer festen Tiefe* h' *aufspaltet, muß* $h' \geq h - \log \frac{T_{\text{seq}}}{T_{\text{atomic}}} + \log n$ *wählen, damit es kein Teilproblem* P *mit* $T(P) > \frac{T_{\text{seq}}}{n}$ *gibt.*

Beweis. Analog zum Beweis von Satz 5.3. □

Dies ist ein deutliches Nachteil der vielen Algorithmen (siehe z.B. KUMAR UND ANANTH (1991) für eine Übersicht), die bis zu $2^{h'}$ Teilprobleme erzeugen und diese an die PEs verteilen. Selbst wenn Erzeugung und Verteilung nach einem ausgeklügelten Schema parallel geschehen, ist der entstehende Aufspaltungsaufwand im allgemeinen nicht tolerierbar. Im Vergleich dazu wird ein Vorteil der empfängerveranlaßten Algorithmen deutlich; diese haben so etwas wie adaptive Granularitätskontrolle. Große Teilprobleme werden dort öfter aufgespalten als kleine, einfach weil sie länger existieren. „Blinde“ Algorithmen funktionieren etwas besser, wenn mehr über die Aufspaltungsfunktion oder Teilproblemgrößen bekannt ist als bei allgemeinen baumförmigen Berechnungen. Für Baumsuchprobleme wird das in Abschnitt 2.3.5 beschriebene α-Spalten häufig betrachtet. Aber auch hier muß die Spalttiefe h' hinreichend groß gewählt werden.

Satz 5.8. *Sei* $x(\alpha) := \frac{1}{\log \frac{1}{1-\alpha}}$. *Im* α*-Spaltmodell muß jeder Lastverteilungsalgorithmus, der das Wurzelproblem bis zu einer festen Tiefe* h' *aufspaltet,* $h' \geq x(\alpha) \log n$ *wählen, damit es kein Teilproblem* P *mit* $T(P) > \frac{T_{\text{seq}}}{n}$ *gibt.*

Beweis. Wird immer im Verhältnis $\alpha : (1-\alpha)$ aufgespalten, so gibt es ein Teilproblem P mit $\text{gen}(P) = h'$ und $T(P) = T_{\text{seq}}(1-\alpha)^{h'}$. Erst für $h' \geq \frac{1}{\log \frac{1}{1-\alpha}} \log n = x(\alpha) \log n$ unterschreitet dieser Wert $\frac{T_{\text{seq}}}{n}$. □

Aus dem Zeitbedarf für das Erzeugen von $2^{h'}$ Teilproblemen ergibt sich damit die folgende Schranke für die parallele Ausführungszeit.

Korollar 5.9. *Soll eine Effizienz in* $\Omega(1)$ *erreicht werden, so gilt für Algorithmen, die den kompletten Spaltbaum bis zu einer festen Tiefe erzeugen, im* α*-Spaltmodell*

$$T_{\text{par}} \in \Omega\left(\frac{T_{\text{seq}}}{n} + T_{\text{atomic}} + d + l + T_{\text{split}} n^{x(\alpha)-1}\right) \quad .$$

Wenn α nicht nahe bei $\frac{1}{2}$ liegt, ergibt sich daraus eine deutlich schlechtere Skalierbarkeit als mit empfängerveranlaßten Algorithmen zu erreichen ist. Insbesondere ist die Analyse in REINEFELD (1994) nicht haltbar, weil angenommen wird, daß es genügt, $O(n)$ Teilprobleme zu erzeugen. Korollar 5.9 deckt sich auch mit der ebendort gemachten experimentellen Beobachtung, daß sehr viele Teilprobleme pro PE erzeugt werden müssen. Der gleiche Punkt muß auch bei den in KUMAR UND ANANTH (1991) und anderen Arbeiten dieser Autoren angestellten Vergleichen bedacht werden.

5.4 Zusammenfassung

Es lassen sich untere Schranken für die Effektivität von Lastverteilungsalgorithmen für baumförmige Berechnungen angeben, die auf sehr einfachen Eigenschaften beruhen, wie „Irgendwo muß die sequentielle Arbeit getan werden.“, „Atomare Teilprobleme sind nicht spaltbar.“, „Um alle PEs zu aktivieren, muß der Netzwerkdurchmesser überbrückt werden.“. Die Sätze 5.2, 5.3, 5.4 und 5.7 beruhen darauf, daß es Berechnungen gibt, bei denen hinreichender Parallelismus nur erreicht wird, wenn viele Aufspaltungen vorgenommen werden. In der Literatur gibt es Ergebnisse, die Vermutungen für noch stärkere Schranken nahelegen: In WU UND KUNG (1991) wird (in etwa) gezeigt, daß alle *deterministischen* Lastverteilungsalgorithmen ebenfalls der Schranke aus Korollar 5.5 gehorchen müssen. Leider scheinen viele interessante Lastverteilungsalgorithmen randomisiert zu sein, so daß eine weitere Verallgemeinerung eine interessante Frage darstellt. Arbeiten über die Online-Einbettung von Bäumen[1)] in Netzwerke wie KAKLAMANIS UND PERSIANO (1994) legen nahe, daß es außerdem unmöglich ist, mit lokaler Kommunikation auszukommen. Leider wird dort die Annahme gemacht, daß jedes PE nur $O(1)$ Baumknoten aufnehmen darf.

Ergebnisse

Die hier erzielten Ergebnisse sind zwar sehr einfach, verschärfen aber immerhin einige der in KUMAR UND ANANTH (1991) angegebenen Schranken.

[1)] D.h. die Einbettung von Bäumen mit a priori nicht bekannter Struktur.

Kapitel 6

Zufälliges Anfragen

In diesem Kapitel wird die in Abschnitt 2.4.2 bereits erwähnte Strategie des zufälligen Anfragens von verschiedenen Seiten beleuchtet. Dabei stellt sich dieser Algorithmus schon in ihren einfachsten Varianten als ein effizientes und robustes Lastverteilungsverfahren heraus. In Abschnitt 6.1 wird mit synchron arbeitenden Varianten begonnen. Diese sind einfach zu verstehen und zu analysieren, haben auf einigen Maschinen praktische Vorteile, und es gibt eine Variante, die sehr sparsam mit der Ressource Zufallsbits umgeht. Abschnitt 6.2 geht auf den asynchronen Algorithmus ein, der für viele heute übliche MIMD-Rechner sehr gut geeignet ist. Abschnitt 6.3 untersucht die Anfangsphase von zufälligem Anfragen anhand eines einfachen Modells und erlaubt dadurch abzuschätzen, inwieweit verbesserte Initialisierungsstrategien zu verbesserter Leistung führen können. Abschnitt 6.4 faßt die Ergebnisse des Kapitels zusammen.

6.1 Synchrones zufälliges Anfragen

Algorithmen, die mit synchronisierten Phasen von sequentieller Berechnung und Kommunikation arbeiten, haben eine Reihe von Vorteilen. Sie sind leicht zu programmieren und zu testen. Auf SIMD-Rechnern aber auch in einigen Programmiermodellen von MIMD-Rechnern (z.B. VALIANT (1994)) kann prinzipiell nur auf diese Weise kommuniziert werden. Oft ist synchronisierte Kommunikation auch schneller als asynchrone Kommunikation, weil sie einfachere Protokolle zuläßt. Selbst die für nachrichtengekoppelte MIMD-Rechner entwickelte Bibliothek MPI bietet kollektive Operationen bisher nur in einer synchronen Form an (SNIR ET AL., 1996). Hier wird auch deshalb mit synchronen Algorithmen begonnen, weil diese einfacher zu analysieren sind.

Ausgangspunkt ist ein in Abschnitt 6.1.1 eingeführter einfacher Algorithmus, der dann in Abschnitt 6.1.2 analysiert wird. In Abschnitt 6.1.3 wird gezeigt, wie durch den Einsatz von Pseudozufallspermutationen die auftretenden Datentransportprobleme vereinfacht werden können. Außerdem kommt diese Variante mit sehr wenig Zufallsbits aus und spart Lastverteilungsphasen ein. In Abschnitt 6.1.4 wird dann diskutiert, wie sich die Auslösung von Lastausgleichsphasen am besten bewerkstelligen läßt.

6.1.1 Der Algorithmus

Abbildung 6.1 zeigt den Pseudocode für die synchrone Variante von zufälligem Anfragen. Zunächst wird PE 0 mit dem Wurzelproblem initialisiert. Alle anderen PEs erhalten zunächst ein leeres Teilproblem. In Abschnitt 6.3 wird sich herausstellen, daß dieses scheinbar naive Vorgehen recht unproblematisch ist, da die Last durch die nachfolgende Lastverteilung schnell auf alle PEs verteilt wird. Trotzdem gibt es geschicktere Ansätze, deren Diskussion aber Abschnitt 8.4.2 vorbehalten ist. Die eigentliche Problemlösung läuft nun in einer einzigen synchron von allen PEs durchlaufenen Schleife ab. Alle PEs verrichten zu Beginn eines Schleifendurchlaufs ein Quantum Δt sequentieller Arbeit, das hinreichend lang sein sollte, damit die folgende Kommunikationsphase nicht allzusehr ins Gewicht fällt. Die Analyse wird Hinweise darauf ergeben, wie Δt zu wählen ist. Nach einer Arbeitsphase wird zunächst gezählt, wieviele PEs arbeitslos sind. (Mit Hilfe einer globalen Summenreduktion.) Solange die Anzahl arbeitsloser PEs unter einer Schwelle m bleibt, wird unmittelbar zur nächsten sequentiellen Arbeitsphase übergegangen. Die eigentliche Lastverteilung besteht darin, daß arbeitslose PEs unabhängig voneinander eine Lastanfrage an ein zufällig gleichverteilt ausgewähltes PE schicken. PEs, die Lastanforderungen erhalten, spalten ihr Teilproblem auf (bei arbeitslosen Anfragenempfängern gegebenenfalls ein leeres Teilproblem) und schicken es an einen beliebigen der Anforderer. Anfragen, die mit dieser „bevorzugten“ Anfrage kollidieren, werden mit leeren Teilproblemen beantwortet. Die Berechnung ist beendet, sobald es nur noch arbeitslose PEs gibt. (Eine Komponente zum Aufsammeln der Ergebnisse ist für baumförmige Berechnungen nicht nötig, kann aber problemlos integriert werden.)

```
var P, P' : Subproblem
P := if i_PE = 0 then P_root else P_∅
loop
    P := work(P, Δt)                          (* do some sequential work *)
    m' := |{i : T(P⟨i⟩) = 0}|                  (* how many idle PEs? *)
    if m' = n then exit loop                  (* all PEs done *)
    elsif m' ≥ m then                         (* many idle PEs *)
        if T(P) = 0 then                      (* I am idle *)
            send a request to a random PE
        if there is an incoming request then
            (P, P') := split(P)
            send P' to one of the requestors
            send empty subproblems to all other requestors
        if T(P) = 0 then receive P            (* collect reply *)
```

Abbildung 6.1: Grundalgorithmus für synchrones zufälliges Anfragen.

6.1.2 Analyse

Dieser Abschnitt ist dem Beweis des folgenden Satzes gewidmet, der zeigt, daß synchrones zufälliges Anfragen zu einer hohen Effizienz führt, wenn $\frac{T_{\text{seq}}}{n}$ groß gegenüber der atomaren Teilproblemgröße und dem h-fachen des Kommunikationsaufwands pro Schleifendurchlauf ist.

Satz 6.1. *Für alle $\varepsilon > 0$ gibt es eine Wahl von Δt und m, so daß für die Ausführungszeit von Algorithmus 6.1 gilt*

$$T_{\text{par}} \preceq (1+\varepsilon)\frac{T_{\text{seq}}}{n} + \tilde{\mathrm{O}}\left(T_{\text{atomic}} + h(T_{\text{rout}}(l) + T_{\text{coll}} + T_{\text{split}})\right) \quad .$$

Der wichtigste Schritt zum Beweis von Satz 6.1 ist zu zeigen, daß es nur wenige Iterationen mit niedriger PE-Auslastung geben kann. Eine genügende Zahl solcher Iterationen führt nämlich mit hoher Wahrscheinlichkeit zu einer vielfachen Aufspaltung aller Teilprobleme und dies impliziert, daß es irgendwann höchstens noch Teilprobleme mit atomarer Größe gibt.

Lemma 6.2. *Sei $m < n$ mit $m \in \Omega(n)$. Dann genügen $\tilde{\mathrm{O}}(h)$ Iterationen von Algorithmus 6.1, an deren Ende mindestens m PEs ein leeres Teilproblem haben, damit es kein Teilproblem P mit $\text{gen}(P) < h$ mehr gibt.*

Für den Beweis wird die folgende auch später noch wichtige Vereinbarung benötigt:

Definition 6.3. Der *Vorfahr* eines Teilproblems P zum Zeitpunkt t ist dasjenige eindeutig bestimmte Teilproblem, aus dem P durch Anwendung der Operationen „work“ und „split“ abgeleitet wird.

Beweis. Sei $\beta > 0$ beliebig. Sei $c > 0$ eine noch näher zu bestimmende Konstante. Eine Iteration sei „rot“ genannt, wenn an ihrem Ende mindestens m PEs ein leeres Teilproblem haben. In jeder roten Iteration seien außerdem genau m PEs mit leerem Teilproblem rot markiert. Sei k die Anzahl der roten Iterationen. Abbildung 6.2 zeigt ein Beispiel für einen Programmablauf und eine mögliche Färbung von Iterationen und PEs. Nach Gleichung (3.3) genügt zu zeigen, daß für einen beliebigen, aber festen PE-Index i und hinreichend große n gilt

$$k \geq ch \implies \mathbf{P}[\text{gen}(P\langle i\rangle) < h] < n^{-\beta-1} \quad .$$

Sei die Indikatorzufallsvariable $X_j := 1$, wenn nach der j-ten roten Iteration der Vorfahr von $P\langle i\rangle$ eine Anfrage von einem roten PE erhält (und Null sonst). Dann ist $\text{gen}(P\langle i\rangle) \geq \sum_{j<k} X_j$. X_j ist genau dann Null, wenn keine der m unabhängigen roten Anfragen PE i erreicht. Die Wahrscheinlichkeit, daß eine einzelne rote Anfrage PE j *nicht* erreicht, ist $1 - 1/n$. Insgesamt gilt

$$\mathbf{P}[X_j = 1] = 1 - (1-1/n)^m \geq 1 - e^{-m/n}$$

(mit Hilfe von Gleichung (3.1)). Wegen $m \in \Omega(n)$ läßt sich diese Wahrscheinlichkeit durch eine Konstante abschätzen, die im folgenden mit γ abgekürzt sei. Außerdem sind die X_j unabhängig, da es nur von den in dieser Iteration gewählten Zufallszahlen abhängt, ob der

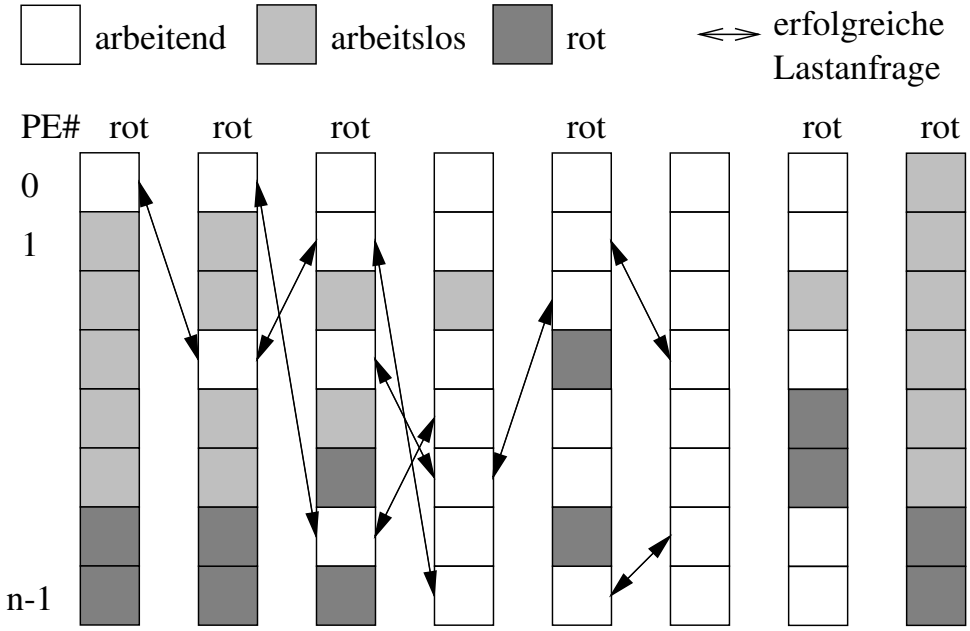

Abbildung 6.2: Beispiel für synchrones zufälliges Anfragen. ($n = 8$, $m = 2$, $k = 6$.)

Vorfahr eine Anfrage von einem roten PE erhält. Nun kann die Chernoff-Schranke (3.6) (mit den Ersetzungen $q \leftarrow \gamma$, $n \leftarrow k$, $\varepsilon \leftarrow 1 - \frac{h}{\gamma k}$) herangezogen werden.

$$\mathbf{P}[\text{gen}(P\langle i\rangle) < h] \leq \mathbf{P}\left[\sum_{j<k} X_j < h\right] \leq e^{-(1-\frac{h}{\gamma k})^2 \gamma k/3} \leq e^{-(1-\frac{1}{\gamma c})^2 \gamma c h/3}$$

für $k \geq ch$. Nach Relation 2.1 gibt es ein c' so daß $h \geq c' \ln n$:

$$\leq n^{-(1-\frac{1}{\gamma c})^2 \gamma c c'/3} = n^{-\frac{1}{3}c'(\gamma c - 2 + \frac{1}{\gamma c})} \leq n^{-\frac{1}{3}c'(\gamma c - 2)}$$

$$\leq n^{-\beta-1} \text{ falls } c \geq \frac{\frac{3(\beta+1)}{c'} + 2}{\gamma} \quad .$$

□

Der zweite tragende Pfeiler des Beweises ist die Beobachtung, daß es auch nur wenige Iterationen mit hoher PE-Auslastung geben kann, da nur eine begrenzte Menge sequentieller Arbeit vorhanden ist.

Lemma 6.4. *Sei $m < n$ mit $m \in \Omega(n)$. Dann gibt es höchstens $\frac{T_{\text{seq}}}{(n-m)\Delta t}$ Iterationen von Algorithmus 6.1, die an ihrem Ende höchstens m PEs mit leerem Teilproblem haben.*

Beweis. Angenommen, es gäbe mehr solche Iterationen. Haben am Ende einer Iteration höchstens m PEs ein leeres Teilproblem, so müssen mindestens $n-m$ PEs während dieser Iteration sequentielle Arbeit im Umfang von jeweils Δt verrichtet haben. Folglich muß spätestens nach der $\left\lceil \frac{T_{\text{seq}}}{(n-m)\Delta t} \right\rceil$-ten solchen Iteration sequentielle Arbeit im Umfang von T_{seq} verrichtet worden sein. Nach der Definition von baumförmigen Berechnungen aus Abschnitt 2.3.1 kann dann keine sequentielle Arbeit und somit auch kein nichtleeres Teilproblem mehr übrig sein. □

Abbildung 6.3 verdeutlicht den komplementären Charakter der Lemmata 6.2 und 6.4. Jede Iteration hat entweder eine hohe oder eine niedrige PE-Auslastung. Bei Iterationen mit hoher Auslastung kann die geleistete Arbeit als Produktivkapital verbucht werden. Bei niedriger Auslastung werden viele Anfragen abgesetzt und die verbleibenden Teilprobleme werden dadurch „zerkleinert“. Der Beweis von Satz 6.1 kann nun komplettiert werden, indem der Zeitaufwand für die einzelnen Komponenten des Algorithmus und eine geeignete Einstellung der Parameter Δt und m bestimmt werden.

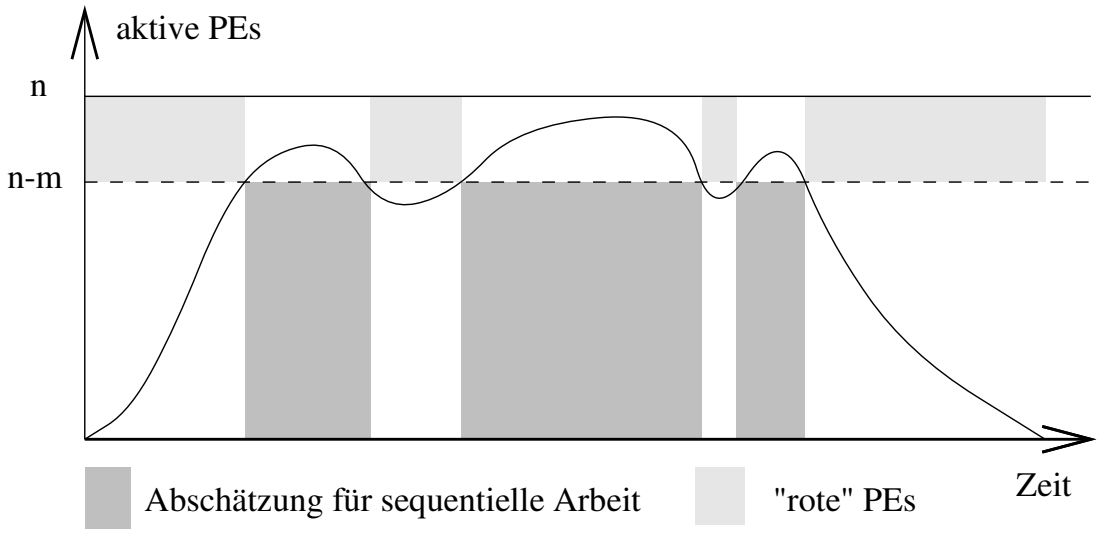

Abbildung 6.3: Prozessorauslastung

Sei $m < n$ und $m \in \Omega(n)$. Außerdem sei eine Konstante c so gewählt, daß die Zeit, die pro Iteration auf Synchronisation, Berechnung von $|\{i : T(P\langle i\rangle) = 0\}|$ und sonstigen Kontrollaufwand verwendet wird, höchstens cT_{coll} beträgt. Es sei der Ansatz $\Delta t := c' \max\{T_{\text{rout}}(l), T_{\text{coll}}, T_{\text{split}}\}$ für den Umfang sequentieller Arbeit pro Iteration gemacht.

Aus Lemma 6.4 folgt, daß es höchstens $\frac{T_{\text{seq}}}{(n-m)\Delta t}$ Iterationen ohne Lastausgleich gibt. Der Zeitbedarf für jede dieser Iterationen ist höchstens $\Delta t + cT_{\text{coll}}$. Multiplikation dieser Größen ergibt eine Schranke für den Zeitaufwand für Iterationen ohne Lastverteilung.

$$\begin{aligned} T_1 &:= \frac{T_{\text{seq}}}{(n-m)\Delta t}(\Delta t + cT_{\text{coll}}) = \frac{T_{\text{seq}}}{n-m}(1 + \frac{cT_{\text{coll}}}{\Delta t}) \\ &= \frac{T_{\text{seq}}}{n-m}\left(1 + \frac{c}{c'}\frac{T_{\text{coll}}}{\max\left\{T_{\text{rout}}(l), T_{\text{coll}}, T_{\text{split}}\right\}}\right) \\ &\leq \frac{T_{\text{seq}}}{n-m}\left(1 + \frac{c}{c'}\right) = \frac{1+\frac{c}{c'}}{1-\frac{m}{n}} \cdot \frac{T_{\text{seq}}}{n} \quad . \end{aligned}$$

Um diese Zeit durch $(1+\varepsilon)\frac{T_{\text{seq}}}{n}$ nach oben abschätzen zu können, genügt es, c' und m so zu wählen, daß $(1+\frac{c}{c'})/(1-\frac{m}{n}) \leq 1+\varepsilon$. Eine Möglichkeit ist z.B. $m = \left\lfloor n\frac{\varepsilon}{2+\varepsilon}\right\rfloor$ und $c' = c\frac{2+\varepsilon}{\varepsilon}$ (und damit $\Delta t = c\frac{2+\varepsilon}{\varepsilon}\max\left\{T_{\text{rout}}(l), T_{\text{coll}}, T_{\text{split}}\right\}$.)

Nach Lemma 6.2 gibt es nach $\tilde{\mathrm{O}}(h)$ Iterationen mit Lastverteilung nur noch Teilprobleme P mit $\text{gen}(P) \geq h$ und damit $T(P) \leq T_{\text{atomic}}$. Diese sind in spätestens $\left\lceil \frac{T_{\text{atomic}}}{\Delta t}\right\rceil \leq 1 + \frac{T_{\text{atomic}}}{\Delta t}$

Iterationen abgearbeitet. Insgesamt kann es also nicht mehr als $\tilde{\mathrm{O}}(h) + \frac{T_{\text{atomic}}}{\Delta t}$ Iterationen mit Lastverteilung geben. Diese Iterationen benötigen einen zusätzlichen Zeitaufwand von $T_{\text{split}} + \tilde{\mathrm{O}}(T_{\text{rout}}(l))$ für das Spalten von Teilproblemen und den Transport von Nachrichten der Länge $\mathrm{O}(l)$. Nach Einsetzen von $\Delta t = c' \max\left\{T_{\text{rout}}(l), T_{\text{coll}}, T_{\text{split}}\right\}$ läßt sich der Gesamtaufwand einer Iteration mit Lastverteilung durch $\tilde{\mathrm{O}}\left(T_{\text{rout}}(l) + T_{\text{coll}} + T_{\text{split}}\right)$ abschätzen. Multiplikation dieser Größen ergibt eine Abschätzung für die Zeit T_2, die für Iterationen mit Lastverteilung benötigt wird.

$$\begin{aligned} T_2 &\in \tilde{\mathrm{O}}\left(\left(h + \frac{T_{\text{atomic}}}{\Delta t}\right)\left(T_{\text{rout}}(l) + T_{\text{coll}} + T_{\text{split}}\right)\right) \\ &= \tilde{\mathrm{O}}\left(h(T_{\text{rout}}(l) + T_{\text{coll}} + T_{\text{split}}) + T_{\text{atomic}} \frac{T_{\text{rout}}(l) + T_{\text{coll}} + T_{\text{split}}}{c' \max\left\{T_{\text{rout}}(l), T_{\text{coll}}, T_{\text{split}}\right\}}\right) \\ &= \tilde{\mathrm{O}}\left(h(T_{\text{rout}}(l) + T_{\text{coll}} + T_{\text{split}}) + T_{\text{atomic}}\right) \end{aligned}$$

Die Aussage von Satz 6.1 folgt nun aus $T_{\text{par}} \leq T_1 + T_2$. ∎

Algorithmus 6.1 ist bereits ein recht guter Lastverteilungsalgorithmus. Zunächst sei aber auf einige Schwächen eingegangen, die in den nächsten Abschnitten beseitigt werden. Besonders bei mehrstufigen Verbindungsnetzwerken und vollständiger Verknüpfung kann die Reduktionsoperation zum Zählen arbeitsloser PEs den Kommunikationsaufwand dominieren, wenn Teilproblembeschreibungen kurz sind. Außerdem führen Kollisionen zwischen Lastanfragen dazu, daß der Kommunikationsaufwand steigt und die Chance für eine erfolgreiche Lastanfrage geschmälert wird.

6.1.3 Pseudozufallspermutationen

Kollisionen zwischen Lastanfragen lassen sich vermeiden, indem in jeder Lastverteilungsphase eine Zufallspermutation $\pi(i)$ verwendet wird. Wird PE i arbeitslos, so richtet es eine Anfrage an PE $\pi(i)$.

Leider ist die Bestimmung einer echten Zufallspermutation mit den Algorithmen aus Abschnitt 4.3.1 so teuer, daß dadurch keine wirkliche Verbesserung zu erwarten ist. Eine nähere Betrachtung der Analyse des Lastverteilungsalgorithmus zeigt jedoch, daß eine echte Zufallspermutation überhaupt nicht nötig ist. Die einzige Anforderung ist, daß für beliebige PEs i und j gilt $\mathbf{P}[\pi(i) = j] = 1/n$. Es genügt also eine Familie $\mathcal{U}$ von Permutationen zu identifizieren, so daß

$$\forall i < n, j < n : |\{\pi \in \mathcal{U} : \pi(i) = j\}| = |\mathcal{U}|/n \ .$$

Dann kann statt einer echten Zufallspermutation ein zufälliges Element aus $\mathcal{U}$ ausgewählt werden. Eine solche „universelle“ Menge von Permutationen ist z.B. die Menge der zyklischen Verschiebungen

$$\mathcal{U}_{\text{shift}} := \{\pi_k \,|\, k \in \mathbb{N}_n, \pi_k(i) = i + k \bmod n\}$$

oder die Menge der exklusiven Oder-Verknüpfungen.

$$\mathcal{U}_{\text{xor}} := \{\pi_k \,|\, k \in \mathbb{N}_n, \pi_k(i) = i \oplus k\} \ .$$

Zum zufälligen Auswählen einer solchen Permutation genügt es, eine einzige Zufallszahl k zu bestimmen und diese allen PEs mitzuteilen.

Selbst wenn der Aufwand für die Bestimmung echter Zufallspermutationen nicht ins Gewicht fiele, wären solche einfachen Pseudozufallspermutationen vorzuziehen. Diese implizieren nämlich Kommunikationsmuster, die durch sehr einfache deterministische Datentransportalgorithmen realisierbar sind.[1)] Außerdem benötigen die beiden oben vorgestellten Klassen von Pseudozufallspermutationen nur $\log n$ Zufallsbits zur Bestimmung einer Permutation. Verglichen mit den $\Theta(n \log n)$ Bits, die für echte Zufallspermutationen oder für die Grundversion von zufälligem Anfragen benötigt werden, ist dies eine exponentielle Verbesserung bezüglich der Ressource Zufallsbits, die von großer theoretischer aber auch praktischer Bedeutung ist (GUPTA ET AL., 1994).

6.1.4 Auslösen einer Lastausgleichsphase

In Algorithmus 6.1 wird eine Lastausgleichsphase immer dann ausgelöst, wenn die Anzahl arbeitsloser PEs eine Schwelle m überschreitet. Dies ist nur eine von vielen Möglichkeiten zur Auslösung einer Lastverteilungsphase. Zum Beispiel kann der Algorithmus die Schwelle m selber bestimmen. In KARYPIS UND KUMAR (1992, 1994); POWLEY ET AL. (1993) geschieht dies zur Laufzeit auf der Basis einer Extrapolation der vergangenen PE-Auslastungsentwicklung und einer Messung von Kosten und Nutzen der letzten Lastausgleichsphase. Es wird berichtet, daß dadurch nicht nur eine bessere Effizienz erreicht wird, als es durch eine feste Schwelle möglich wäre, sondern es wird auch ein nicht ganz leicht einzustellender Parameter eingespart. Diese adaptiven Verfahren funktionieren dann besonders gut, wenn die PE-Auslastung eine relativ „glatte“ und damit leicht extrapolierbare Entwicklung aufweist und eine häufige Messung der PE-Auslastung nicht zu teuer ist. In einer solchen Situation ist zufälliges Anfragen allerdings nicht unbedingt das beste bekannte Verfahren. Die in Abschnitt 2.4.2 bereits erwähnten, auf Präfixsummen beruhenden Algorithmen erreichen dann eine bessere PE-Auslastung.

Bei der Lösung sehr unregelmäßiger, relativ feinkörniger Probleme, die an der Grenze dessen liegen, was auf einer gegebenen Maschine noch lösbar ist, ergeben sich aber andere Anforderungen. Hier ist eine Effizienz von z.B. 95 % ohnehin nicht erreichbar. Es geht darum, die Lastverteilungsphasen möglichst einfach zu halten, um sie so oft wie möglich einsetzen zu können. In dieser Beziehung ist zufälliges Anfragen mit Pseudozufallspermutationen wohl kaum zu schlagen. Wenn eine Lastverteilungsphase aber relativ billig ist und ohnehin sehr oft eingesetzt werden muß, kann auf das Zählen arbeitsloser PEs auch ganz verzichtet werden. Statt dessen wird dann einfach in festen Zeitabständen von Δt eine Lastverteilung ausgelöst. In den Abschnitten A.2.5 und A.6.1 wird auf ein Beispiel dazu eingegangen. Es ergibt sich die bisher einfachste Variante von zufälligem Anfragen, die deshalb in Abbildung 6.4 noch einmal im Zusammenhang dargestellt ist.

Für diesen Algorithmus gilt der folgende Satz:

[1)] Zum Beispiel benötigt eine zyklische Verschiebung auf der Maspar MP-1 ca. 2.3 mal weniger Zeit als eine Zufallspermutation.

var P, P' : Subproblem
$P :=$ **if** $i_{\mathrm{PE}} = 0$ **then** P_{root} **else** $P_\emptyset$
while not finished (* e.g. look for busy PEs from time to time *)
    $P := \mathrm{work}(P, \Delta t)$ (* do some sequential work *)
    select a global value $0 \leq s < n$ uniformly at random
    if $T(P\langle i_{\mathrm{PE}} - s \bmod n\rangle) = 0$ **then**
        $(P, P\langle i_{\mathrm{PE}} - s \bmod n\rangle) := \mathrm{split}(P)$

Abbildung 6.4: Synchrones zufälliges Anfragen mit zyklischer Verschiebung und periodischer Lastverteilung.

Satz 6.5. *Für alle* $\varepsilon > 0$ *gibt es eine Wahl von* Δt*, so daß für die Ausführungszeit von Algorithmus 6.4 gilt*

$$T_{\mathrm{par}} \preceq (1+\varepsilon)\frac{T_{\mathrm{seq}}}{n} + \mathrm{O}(T_{\mathrm{atomic}}) + \mathrm{O}\left(\frac{nl}{b} + d + T_{\mathrm{split}}\right)\tilde{\mathrm{O}}(h) \quad .$$

Es werden außerdem nur $\tilde{\mathrm{O}}(h \log n)$ *Zufallsbits benötigt.*

Beweis. Weitgehend analog zu Beweis von Satz 6.1 Die Iterationen können auch ohne eine Messung der Anzahl arbeitsloser PEs in solche mit mehr und solche mit weniger als m arbeitslosen PEs eingeteilt werden. Es muß nun in jeder Iteration der Aufwand für die Lastverteilung mitgerechnet werden. Außerdem muß gelegentlich eine Oder-Reduktion stattfinden, um festzustellen, ob es noch arbeitende PEs gibt. Dies kann aber so selten geschehen, daß es asymptotisch nicht ins Gewicht fällt. Die untere Schranke γ für die Wahrscheinlichkeit, daß ein PE in einer Lastverteilungsphase eine Anfrage erhält, ist nun nicht mehr $\gamma = 1 - e^{-m/n}$ sondern der größere Wert m/n. Es werden also weniger Lastverteilungsiterationen benötigt. Wichtig ist, daß es auf allen betrachteten Verbindungsnetzwerken einfache deterministische Algorithmen für die Durchführung einer zyklischen Verschiebung gibt. Dies läßt sich in Zeit $\mathrm{O}\left(\frac{nl}{b} + d\right)$ realisieren. □

Ein Vergleich mit der unteren Schranke aus Satz 5.3 zeigt, daß Algorithmus 6.4 für $T_{\mathrm{split}} \in \Omega\left(\frac{nl}{b} + d\right)$ oder $T_{\mathrm{atomic}} \in \Omega\left(\left(\frac{nl}{b} + d + T_{\mathrm{split}}\right)h\right)$ asymptotisch optimal ist. Aus diesen Beziehungen läßt sich außerdem ablesen, daß es sinnvoll ist, einen Aufwand bis zu $\mathrm{O}\left(\frac{nl}{b} + d\right)$ in die Spaltoperation zu stecken, wenn dadurch h verkleinert wird. Auch eine Vergrößerung von T_{atomic} bis zu $\Omega\left(\left(\frac{nl}{b} + d + T_{\mathrm{split}}\right)h\right)$ kann sinnvoll sein, wenn sich dadurch andere Vorteile ergeben. Bezüglich der Anzahl notwendiger Kommunikationen ist der Algorithmus asymptotisch optimal unter allen deterministischen und allen empfängerveranlaßten Lastverteilungsverfahren. Bei Maschinen mit hoher Bisektionsbreite $b \in \Omega(n)$ (z.B. vollständige Verknüpfung, mehrstufige Verbindungsnetzwerke, Hyperwürfel) ergibt sich daraus ein optimaler Kommunikationsaufwand, wenn $l \in \Omega(d)$. Informell ausgedrückt ist zufälliges Anfragen dann optimal, wenn die Tatsache, daß Lastaufspaltungen mit globaler Kommunikation verknüpft sind, nicht ins Gewicht fällt.

6.2 Asynchrones zufälliges Anfragen

Bei synchronem zufälligen Anfragen stört vor allem, daß arbeitslose PEs auf die nächste Lastverteilungsphase warten müssen, bis sie eine Chance bekommen, neue Arbeit zu erhalten. Vor allem, wenn eine Effizienz sehr nahe bei 1 angestrebt wird, ist das problematisch, da dann Δt sehr groß gewählt werden muß. Gerade wenn die PE-Auslastung gut ist, wäre es viel besser, die Anfrage sofort abzuschicken. Es herrscht dann wenig Verkehr im Netzwerk und die Anfrage wird in den allermeisten Fällen Erfolg haben. Auf lose gekoppelten MIMD-Rechnern ist eine Synchronisation der PEs außerdem eine recht aufwendige Operation. Deshalb wird nun die asynchrone Variante näher untersucht. In Abschnitt 6.2.1 wird der Algorithmus vorgestellt. In Abschnitt 6.2.2 wird die erwartete Ausführungszeit abgeschätzt. Abschnitt 6.2.3 zeigt, welche Modifikationen vorzunehmen sind, damit das Ergebnis auch mit hoher Wahrscheinlichkeit gilt.

6.2.1 Der Algorithmus

```
var P, P' : Subproblem
P := if i_PE = 0 then P_root else P_∅
while no global termination yet do
    if T(P) = 0 then                                   (* I need work *)
        send a request to a PE chosen uniformly at random from ℕ_n
    else
        P := work(P, Δt)                               (* do some sequential work *)
    if there is an incoming message M then
        if M is a request from PE j then
            (P, P') := split(P)
            send P' to PE j
        else                                           (* M ist the response to the last work request *)
            P := M
```

Abbildung 6.5: Grundalgorithmus für asynchrones zufälliges Anfragen.

Der in Abbildung 6.5 dargestellte Grundalgorithmus für asynchrones zufälliges Anfragen ähnelt stark dem synchronen Algorithmus aus Abbildung 6.1. Es werden deshalb nur die Unterschiede dargestellt. Die Hauptschleife wird asynchron durchlaufen, und es wird auch nicht mehr gezählt, wie viele PEs arbeitslos sind. Ein arbeitsloses PE sendet in jedem Fall eine Lastanfrage an einen zufälligen Partner und wartet auf Antwort. Beschäftigte PE unterbrechen ihre Arbeit periodisch um *eine* gegebenfalls angekommene Lastanfrage zu beantworten. Im Hintergrund[2)] läuft währenddessen ein Terminierungserkennungsalgorithmus, z.B. das in Abschnitt 4.1 vorgestellte Doppelzählverfahren. Dabei dürfen Lastanfragen und Nachrichten mit leeren Teilproblemen nicht mitgezählt werden, da solche Nachrichten i. allg. ständig un-

[2)] Eine explizite Behandlung mit nur einem Prozeß pro PE ist ebenfalls kein Problem, würde hier aber die Darstellung unnötig verkomplizieren.

terwegs sind, auch wenn alle PEs arbeitslos sind. Der asynchrone Algorithmus enthält mit Δt nur einen einzigen geeignet einzustellenden Parameter.

6.2.2 Analyse im Mittel

Satz 6.6. *Für die parallele Ausführungszeit von asynchronem zufälligem Anfragen bei geeigneter Wahl von Δt gilt*

$$\mathbf{E}T_{\text{par}} \preceq (1+\varepsilon)\frac{T_{\text{seq}}}{n} + \mathrm{O}\big(T_{\text{atomic}} + h(T_{\text{rout}}(l) + T_{\text{split}})\big) \quad .$$

T_{rout} ist als der mittlere Aufwand für das Versenden einer Nachricht der Länge $\mathrm{O}(l)$ bei zufälliger, asynchroner Kommunikation zu verstehen. Wie schon in Abschnitt 2.2.2 diskutiert, kann es nicht Aufgabe dieser Arbeit sein zu untersuchen, inwieweit sich dieser Aufwand vom Aufwand für synchronisierte Kommunikation unterscheidet. Es sei aber erwähnt, daß T_{rout} durchaus eine Zufallsvariable sein kann. Mit Hilfe der Kettenregel 3.9 können die Aussagen, die hier nur für einen nicht näher spezifizierten Parameter gemacht werden, jeweils für Zufallsvariablen verallgemeinert werden.

Der Beweis ähnelt insofern der Analyse des synchronen Algorithmus, als daß die Gesamtrechenzeit in produktive sequentielle Arbeit und Lastverteilungsaufwand eingeteilt wird und für beide Aufwandsarten Schranken hergeleitet werden. Anstelle von Schleifeniterationen, wird nun aber jede einzelne Operation jedes einzelnen PEs einzeln klassifiziert. Außerdem kommt eine zusätzliche Quelle von Aufwand hinzu. Anfragen können sich an PEs stauen und dadurch entsteht ein zusätzlicher Warteaufwand für die auf Antwort wartenden arbeitslosen PEs. Dieser Aufwand ist auch die einzige Größe, die zunächst nur im Mittel abgeschätzt wird. Begonnen sei mit einigen grundlegenden Eigenschaften von Anfragen.

Lemma 6.7. *Für die gesamte aktive Bearbeitung einer Anfrage fällt ein Gesamtaufwand von höchstens $T_{\text{split}} + \mathrm{O}(T_{\text{rout}}(l))$ an.*

Beweis. Eine Anfrage löst maximal eine Teilproblemaufspaltung aus. Der Aufwand für Verschicken, Empfangen (und bei Softwareroutern Weiterleiten) von Anfrage und Antwort liegt in $\mathrm{O}(T_{\text{rout}}(l))$. □

Selbst für die Beantwortung einer einzelnen Anfrage kommt eine Wartezeit von bis zu Δt hinzu:

Lemma 6.8. *Die Zeit für die Beantwortung einer einzigen Anfrage ist höchstens in*

$$\Delta t + T_{\text{split}} + \mathrm{O}(T_{\text{rout}}(l))$$

Anfragen können sich außerdem an PEs stauen. Die Warteschlangen können im Prinzip sogar ziemlich lang werden. Dennoch gilt:

Lemma 6.9. *Die erwartete Zeit, die von der Ankunft einer Anfrage bis zum Abschicken der Antwort vergeht, ist in $\mathrm{O}\big(\Delta t + T_{\text{split}} + T_{\text{rout}}(l)\big)$.*

Beweis. Es sind maximal n Anfragen gleichzeitig unterwegs. Folglich können auch nicht mehr auf Abfertigung warten. Kommt nun eine Anfrage an einer zufälligen PE an, so ist die erwartete Länge der Warteschlange dort

$$\sum_{i<n} \frac{\text{„Schlangenlänge an PE } i\text{“}}{n} \leq 1 \quad .$$

□

In der Analyse des synchronen Algorithmus war es wichtig, daß jeder Vorfahr eines Teilproblems durch jede Anfrage aufgespalten werden konnte. Nun kann es passieren, daß ein Teilproblem nicht „erreichbar“ ist, z.B. weil es sich gerade im Kommunikationsnetz befindet. Um diese „Totzeiten“ zu beschränken, sind einige genauere Begriffsbildungen angebracht. Neben den schon in Definition 6.3 eingeführten Begriff „Vorfahr“ erhalten nun auch die Wörter „Totzeit“, „erreichen“ und „ankommen“ eine spezielle technische Bedeutung.

Definition 6.10. Wird ein Teilproblem von einer Lastanfrage aufgespalten und zu einem anderen PE transportiert so kann es während eines bestimmten Zeitraums von keiner anderen Lastanfrage erreicht werden. T_{tot} bezeichne eine obere Schranke für die Dauer dieses Zeitraums. (Wird ein Teilproblem von k Anfragen aufgespalten, so kann die Gesamttotzeit kT_{tot} betragen.) Eine Lastanfrage *erreicht* ein Teilproblem P zum Zeitpunkt t, wenn sie zu einem Zeitpunkt t *ankommt*, d.h., in den Nachrichtenpuffer eines PE aufgenommen wird und irgendwann danach zur Aufspaltung des Vorfahren von P führt. Dabei wird in in einer (gedachten) globalen Zeit gerechnet.

Lemma 6.11. $T_{\text{tot}} \leq \Delta t + T_{\text{split}} + T_{\text{rout}}(l)$.

Beweis. Sei P ein Teilproblem, das zum Zeitpunkt t von einer Anfrage A auf PE i erreicht wird. Außerdem gebe es k weitere Anfragen, die vor A im Nachrichtenpuffer stehen, und folglich ebenfalls P erreichen. Falls der Vorfahr von P aufgrund von A zu einem anderen PE j transportiert wird, so ist P bis zu dem Zeitpunkt nicht erreichbar, zu dem es in den Nachrichtenpuffer von PE j aufgenommen wird. Schlimmstenfalls vergeht bis dahin eine Zeit von $(k+1)(\Delta t + T_{\text{split}} + T_{\text{rout}}(l))$. Das ist dann der Fall, wenn „work“ für den Vorfahr von P gerade aufgerufen wurde. Dann vergeht eine Zeit Δt bis der Lastverteiler wieder zum Zug kommt. Anschließend wird der Vorfahr von P unter einem Zeitaufwand von T_{split} aufgespalten. Anschließend wird ein abgespaltenes Teilproblem abgeschickt, das nach einer Zeit $T_{\text{rout}}(l)$ in den Nachrichtenpuffer des Empfängers aufgenommen wird. (Für $k \geq 1$ ist dies nicht der Vorfahr von P, sonst würde A das Teilproblem P nicht erreichen.) Dieser Vorgang wiederholt sich $k+1$ mal. Danach ist P auf PE j erreichbar. Diese Gesamttotzeit kann auf die $k+1$ Anfragen, die P erreicht haben, aufgeteilt werden, so daß für jede ein Zeitaufwand von höchstens $\Delta t + T_{\text{split}} + T_{\text{rout}}(l)$ anfällt. □

Die Grundidee ist nun, festzustellen, wieviele Anfragen benötigt werden, um jedes Teilproblem mit hoher Wahrscheinlichkeit h mal zu spalten. Wegen der Totzeiten ist diese Frage aber noch ungenau gestellt. Anfragen, die während einer Totzeit eines Teilproblems ankommen, sind für dieses Teilproblem „verloren“. Deshalb wird nun eine Teilmenge der insgesamt abgearbeiteten Anfragen betrachtet, die die Eigenschaft hat, einigermaßen „gleichmäßig“ über die Zeit verteilt zu sein.

Definition 6.12. Eine Anfrage darf *rot* gefärbt werden, wenn in einem Zeitraum von T_{tot} nach ihrer Ankunft höchstens n andere rote Anfragen ankommen.

Lemma 6.13. *Für jedes $\beta > 0$ gibt es eine Konstante $c > 0$, so daß nach Bearbeitung von cnh roten Anfragen gilt $\mathbf{P}[\exists i : \text{gen}(P\langle i\rangle) < h] \leq n^{-\beta}$ (für hinreichend große n).*

Beweis. Es genügt zu zeigen, daß für einen beliebigen aber festen PE-Index i, und hinreichend große n gilt $\mathbf{P}[\text{gen}(P\langle i\rangle) < h] < n^{-\beta-1}$ (Der Term „-1“ erklärt sich wieder aus Abschätzung (3.3). $\text{gen}(P\langle i\rangle)$ läßt sich durch die Anzahl roter Anfragen abschätzen, die $P\langle i\rangle$ erreichen. Ungefärbte Anfragen können o.B.d.A. außer acht gelassen werden. Zwar kann es passieren, daß eine ungefärbte Anfrage $P\langle i\rangle$ erreicht und verhindert, daß eine oder mehrere rote Anfragen $P\langle i\rangle$ erreichen, aber diese Aufspaltung wird dann der ersten folgenden roten Anfrage zugerechnet und die für diese rote Anfrage berücksichtigte Totzeit ist hinreichend lang, um das Nichterreichen durch die folgenden roten Anfragen zu erklären.

Es wird nun durch kombinatorische Überlegungen gezeigt, daß

$$\sum_{k<h} \mathbf{P}[P\langle i\rangle \text{ wird von } k \text{ roten Anfragen erreicht}] \leq n^{-\beta-1} \quad .$$

Es gibt $\binom{chn}{k}$ Möglichkeiten, k rote Anfragen auszuwählen, die $P\langle i\rangle$ erreichen sollen. Die Wahrscheinlichkeit, daß diese alle PE i ansteuern, ist n^{-k}. Da in die Totzeit nach einer roten Anfrage höchstens n andere rote Anfragen fallen, bleiben $chn - kn$ rote Anfragen, die $P\langle i\rangle$ nicht erreichen. Die Wahrscheinlichkeit dafür beträgt $(1-1/n)^{chn-kn} \leq e^{-(ch-k)}$. Insgesamt ergibt sich

$$P_k := \mathbf{P}[P\langle i\rangle \text{ wird von } k \text{ roten Anfragen erreicht}] \leq \binom{chn}{k} n^{-k} e^{-(ch-k)}$$

und mit der Stirling-Abschätzung $\binom{m}{k} \leq (me/k)^k$

$$\leq \left(\frac{chne}{k}\right)^k n^{-k} e^{-(ch-k)} = \left(\frac{che^2}{k}\right)^k e^{-ch} \quad .$$

Ein Ausdruck der Form $(\alpha/k)^k$ wird maximiert für $k = \alpha/e$. Da aber (für $c > \frac{1}{e}$) $k < \frac{che^2}{e}$, ist der k-abhängige Teil der obigen Formel monoton steigend und läßt sich nach oben abschätzen, indem $k = h$ gesetzt wird.

$$P_k \leq \left(ce^2\right)^h e^{-ch} = e^{-h(c-\ln c-2)} \quad .$$

Nun läßt sich $\mathbf{P}[\text{gen}(P\langle i\rangle) < h]$ durch das h-fache des so berechneten Ausdrucks abschätzen.

$$\begin{aligned}\mathbf{P}[\text{gen}(P\langle i\rangle) < h] &\leq he^{-h(c-\ln c-2)} = e^{-h\left(c-\ln c-2-\frac{\ln h}{h}\right)} \\ &\leq e^{-h\left(c-\ln c-2-\frac{1}{e}\right)}\end{aligned}$$

Nach Relation 2.1 gibt es ein c' so daß $h \geq c' \ln n$:

$$\leq n^{-c'\left(c-\ln c-2-\frac{1}{e}\right)} \leq n^{-\beta-1}$$

für ein geeignetes c und hinreichend große n. □

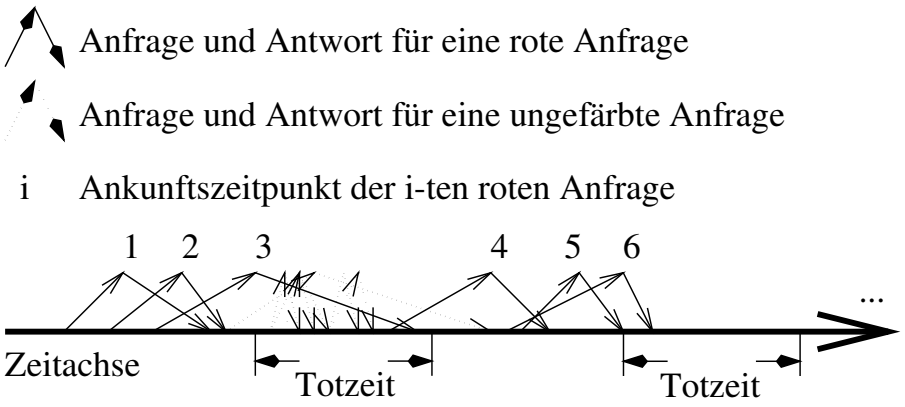

Abbildung 6.6: Beispiel für die Färbung von Anfragen ($n = 3$).

Nun wird der für alle Anfragen aufzuwendende Aufwand abgeschätzt, damit mindestens *cnh* rote Anfragen darunter sind.

Lemma 6.14. *Sei $c > 0$ eine Konstante. Anfragen lassen sich so färben, daß ein mittlerer Gesamtaufwand für alle Anfragen von $\mathrm{O}\big(hn(\Delta t + T_{\text{split}} + T_{\text{rout}}(l))\big)$ genügt, um chn rote Lastanfragen abzuwickeln.*

Beweis. A_i ($i \in \mathbb{N}_m$) seien die insgesamt abgewickelten Lastanfragen. $t(A_i)$ sei der Ankunftszeitpunkt von A_i. Die Anfragen seien nach aufsteigendem Ankunftszeitpunkt geordnet. Es werden jeweils n aufeinanderfolgende Anfragen rot gefärbt und alle in einem Abstand T_{tot} folgenden Anfragen werden nicht gefärbt, usw. Abbildung 6.6 verdeutlicht dieses Schema an einem einfachen Beispiel für $n = 3$. Die Zeitachse läßt sich damit in rote und ungefärbte Intervalle aufteilen. Da maximal n Anfragen gleichzeitig unterwegs sein können, kommen auf jeweils n rote Anfragen höchstens $2n$ ungefärbte Anfragen, deren Bearbeitung in rote Intervalle hereinreicht. Der Aufwand für die Abwicklung von n roten Anfragen läßt sich also durch nT_{tot} plus den Aufwand für die Abwicklung von $3n$ Abfragen nach oben abschätzen. Der Aufwand dafür ergibt sich aus den Lemmata 6.8 und 6.9. □

Aus der Kombination von Lemma 6.14 und Lemma 6.13 ergibt sich damit eine Abschätzung für den Kommunikationsaufwand des Algorithmus. (Hier läßt sich auch problemlos der Aufwand für die Terminierungserkennung unterbringen, die, wie man leicht sieht, allerhöchstens $\mathrm{O}(n)$ kurze Nachrichten für je n Anfragen verschickt.)

Lemma 6.15. *Der Erwartungswert des Gesamtaufwands für Kommunizieren, Spalten und Warten, bis zu dem Zeitpunkt, an dem es kein Teilproblem mit $\mathrm{gen}(P) < h$ mehr gibt, ist in $\mathrm{O}(hn)\,(\Delta t + T_{\text{split}} + T_{\text{rout}}(l))$.*

Die Bestimmung der mit sequentieller Arbeit verbrachten Zeit ist wieder relativ einfach.

Lemma 6.16. *Es gibt eine Konstante G, so daß falls $\Delta t > G$, alle PEs insgesamt höchstens $(1+\varepsilon)T_{\text{seq}}$ Zeiteinheiten mit der sequentiellen Bearbeitung von Teilproblemen und erfolglosem Nachsehen im Nachrichteneingangspuffer verbringen.*

Beweis. Analog dem Beweis von Lemma 6.4 für den synchronen Algorithmus. Δt muß nur die Bedingung erfüllen, daß der Aufwand für eine Iteration der Hauptschleife von Algorithmus 6.5, in der keine Nachricht ankommt, höchstens $(1+\varepsilon)\Delta t$ beträgt. Dieser Aufwand hängt nicht von n ab, da nur der konstante Aufwand für die Schleifenkontrolle und ein erfolgloses versuchendes Empfangen hinzukommt. In Iterationen, in denen der Nachrichtenempfang nicht erfolglos ist oder in denen an Teilproblemen mit $T(P) < \Delta t$ gearbeitet wird, kann der Kommunikationsaufwand der Abwicklung von Lastanfragen zugerechnet werden. □

Schließlich muß noch sichergestellt werden, daß atomare Teilprobleme schnell genug „beseitigt" werden und die Terminierungserkennung schnell durchgeführt wird.

Lemma 6.17. *Falls* $\Delta t \in \Omega\left(\min(\frac{T_{\text{atomic}}}{h}, T_{\text{rout}}(l) + T_{\text{split}})\right)$ *und* $\text{gen}(P\langle i\rangle) \geq h$ *für alle* $i \in \mathbb{N}_n$, *so ist die verbleibende Ausführungszeit in*

$$\tilde{\mathrm{O}}\left(T_{\text{atomic}} + T_{\text{coll}} + h(T_{\text{split}} + T_{\text{rout}}(l))\right) \quad .$$

Beweis. Für alle noch vorhandenen Teilprobleme P gelte $\text{gen}(P) \geq h$ und folglich $T(P) \leq T_{\text{atomic}}$. Ist $\frac{T_{\text{atomic}}}{h} \in \mathrm{O}\big(T_{\text{rout}}(l) + T_{\text{split}}\big)$, so genügen $\mathrm{O}(h)$ Iterationen mit Kosten in $\tilde{\mathrm{O}}\left(T_{\text{split}} + T_{\text{rout}}(l)\right)$, um alle Teilprobleme abzuarbeiten. Andernfalls verbringt ein beschäftigtes PE mindestens einen konstanten Anteil seiner Zeit mit produktiver Arbeit, selbst wenn ständig Lastanforderungen kommen.[3)] Folglich ist nach einer Zeitspanne in $\mathrm{O}(T_{\text{atomic}})$ überhaupt kein nichtleeres Teilproblem mehr vorhanden. Nach einer Zeit in $\mathrm{O}(T_{\text{coll}})$ kann[4)] dies die Terminierungserkennung auch feststellen. Danach werden keine neuen Anfragen mehr abgeschickt, und es gilt nur noch die Kommunikationspuffer zu leeren. Dies kann aber nicht länger dauern, als vorher der Aufbau der Warteschlangen. □

Nach dem Beweis der obigen Lemmata ist es nun leicht, den Beweis von Satz 6.6 zu vervollständigen. Da jede Operation von Algorithmus 6.5 sich entweder der Arbeit an einem nichtleeren Teilproblem im Sinne von Lemma 6.16 oder der Bearbeitung einer Lastanfrage im Sinne von Lemma 6.15 zuordnen läßt, kann es nach einer erwarteten Zeit in $(1+\varepsilon)\frac{T_{\text{seq}}}{n} + \mathrm{O}\big(h(T_{\text{rout}}(l) + T_{\text{split}})\big)$ nur noch Teilprobleme mit $\text{gen}(P) \geq h$ geben. Nach Lemma 6.17 genügt dann eine zusätzliche Zeit in $\mathrm{O}\big(T_{\text{atomic}} + T_{\text{coll}} + h(T_{\text{split}} + T_{\text{rout}}(l))\big)$, um verbliebene Teilprobleme atomarer Größe abzuarbeiten und dies auch festzustellen. Wird irgendein

$$\Delta t \in \mathrm{O}\big(T_{\text{rout}}(l) + T_{\text{split}}\big) \cap \Omega\left(\min(\frac{T_{\text{atomic}}}{h}, T_{\text{rout}}(l) + T_{\text{split}})\right) \tag{6.1}$$

gewählt, das größer als die Konstante aus Lemma 6.16 ist, so ergibt sich eine erwartete Gesamtausführungszeit in

$$(1+\varepsilon)\frac{T_{\text{seq}}}{n} + \mathrm{O}\big(h(T_{\text{rout}}(l) + T_{\text{split}}) + T_{\text{atomic}} + T_{\text{coll}}\big) \quad .$$

[3)] Wenn das benutzte System ein Kommunikationsprotokoll hat, bei dem auf der Senderseite nur ein Aufwand von $\mathrm{O}(l)$ für eine asynchrone Sendeoperation anfällt, genügt sogar $\Delta t \in \Omega\left(\min(\frac{T_{\text{atomic}}}{h}, T_{\text{split}} + l)\right)$.

[4)] Ob dies in einer praktischen Implementierung auch geschieht, hängt davon ab, inwieweit die für das Terminierungserkennungsprotokoll nötigen Nachrichten von den weiterhin verschickten Anfragen und den noch vorhandenen Warteschlangen aufgehalten werden. Der insgesamt anfallende Aufwand wird aber nicht größer sein als die vorher für Kommunikation aufgewandte Zeit.

Wegen $T_{\text{coll}} \in \mathrm{O}(hT_{\text{rout}}(l))$ vereinfacht sich dies zu

$$= (1+\varepsilon)\frac{T_{\text{seq}}}{n} + \mathrm{O}\big(T_{\text{atomic}} + h(T_{\text{rout}}(l) + T_{\text{split}})\big) \quad .$$

■

Asynchrones zufälliges Anfragen ist asymptotisch mindestens so effizient wie die synchrone Variante. Rechnet man für eine synchrone Arbeitsweise die Kosten für eine Synchronisation in jedem Zyklus mit T_{coll}, so ist die asynchrone Variante bei vollständiger Verknüpfung und mehrstufigen Verbindungsnetzwerken auch asymptotisch effizienter. Der Parameter Δt ist in der asynchronen Variante sehr leicht einzustellen. Vor allem im (häufigen) Fall, daß $\frac{T_{\text{atomic}}}{h} \ll T_{\text{rout}}(l) + T_{\text{split}}$, zeigt Gleichung 6.1, daß es aus einem weiten Bereich gewählt werden kann, ohne die Leistung des Algorithmus nachhaltig zu beeinflussen. Meist läßt sich sogar ein Δt wählen, das für alle n gut funktioniert.

6.2.3 Analyse mit hoher Wahrscheinlichkeit

Im Gegensatz zu den meisten anderen Algorithmenanalysen in dieser Arbeit, wird in Satz 6.6 nur der Erwartungswert der parallelen Ausführungszeit abgeschätzt. Im folgenden soll gezeigt werden wie Analyse und Algorithmus so abgeändert werden können, daß die Aussage auch mit hoher Wahrscheinlichkeit gilt.

Satz 6.18. *Für die parallele Ausführungszeit von asynchronem zufälligem Anfragen gilt bei geeigneter Wahl von* Δt

$$T_{\text{par}} \preceq (1+\varepsilon)\frac{T_{\text{seq}}}{n} + \tilde{\mathrm{O}}\left(T_{\text{atomic}} + h(T_{\text{rout}}(l) + T_{\text{split}})\right),$$

wenn eine der folgenden Bedingungen erfüllt ist:

- $h \in \Omega(n \log n)$
- *Warteschlangenlängen in* $\Omega(\sqrt{n})$ *werden durch algorithmische Maßnahmen verhindert.*

Für den Beweis genügt es zu zeigen, daß die für $\mathrm{O}(hn)$ rote Anfragen aufgewandte Gesamtwartezeit in $\tilde{\mathrm{O}}\left(nh(\Delta t + T_{\text{split}} + T_{\text{rout}}(l))\right)$ ist. Alles andere folgt analog zum Beweis von Satz 6.6. Dafür wiederum genügt es zu zeigen, daß die Summe der angetroffenen Warteschlangenlängen für $\mathrm{O}(hn)$ (nicht notwendigerweise rote) Anfragen in $\tilde{\mathrm{O}}(hn)$ ist (siehe auch Lemmata 6.8 und 6.9). Beide in Satz 6.18 genannten Fälle lassen sich auf folgendes Kriterium zurückführen:

Lemma 6.19. *Gibt es eine obere Schranke* $\bar{q}$ *für die maximale Länge einer Warteschlange und gilt* $\bar{q} \in \mathrm{O}\left(\sqrt{\frac{hn}{\log n}}\right)$*, so ist die Summe aller von* $\mathrm{O}(hn)$ *Anfragen angetroffenen Warteschlangenlängen in* $\tilde{\mathrm{O}}(hn)$.

Beweis. Sei $\beta > 0$ beliebig. Sei $m \leq ahn$ die Anzahl betrachteter Anfragen und $\bar{q} \leq \sqrt{\frac{bhn}{\log n}}$ eine Schranke für die Warteschlangenlängen (mit geeigneten Konstanten a, b). Die Zufallsvariable T_i $(1 \leq i \leq m)$ bezeichne die Nummer des PE, auf dem Anfrage i ankommt. Die Zufallsvariable Y_i $(1 \leq i \leq m)$ bezeichne die von Anfrage i vorgefundene Warteschlangenlänge. Sei $X := \sum_{i=1}^{m} Y_i$ und $X_i := \mathbf{E}[X \mid T_1, \ldots, T_i]$ $(0 \leq i \leq m)$ die bedingte Erwartung von X in Abhängigkeit von T_1 bis T_i. Nach MOTWANI UND RAGHAVAN (1995) Satz 4.13 und analog Beispiel 4.12 ist $X_0, \ldots, X_m$ eine Martingalfolge. Es gilt außerdem

$$X_i = \sum_{j=1}^{i} Y_j + \mathbf{E}\sum_{j=i+1}^{m} Y_j = \sum_{j=1}^{i} Y_j + \sum_{j=i+1}^{m} \mathbf{E}Y_j$$

und damit

$$|X_{i+1} - X_i| = |Y_{i+1} - \mathbf{E}Y_{i+1}| \leq \bar{q} \ .$$

Sei c eine geeignet zu wählende Konstante. Mit Hilfe von Azumas Ungleichung (Satz 3.4) folgt nun

$$\begin{aligned}\mathbf{P}\left[|X - \mathbf{E}X| \geq chn\right] = \mathbf{P}\left[|X_m - X_0| \geq chn\right] &\leq 2e^{-\frac{(chn)^2}{2m\bar{q}^2}} \\ &\leq 2e^{-\frac{(chn)^2}{2(ahn)(bhn/\log n)}} = 2e^{-\frac{c^2 \log n}{2ab}} = 2n^{-\frac{c^2}{2ab}} \\ &\leq n^{-\beta} \text{ für } c > \sqrt{\frac{\beta}{2ab}} \text{ und hinreichend große } n \ .\end{aligned}$$

Analog zum Beweis von Lemma 6.9 ist außerdem $\mathbf{E}X \leq ahn$. Insgesamt ist also die Summe der angetroffenen Schlangenlängen X in $\tilde{\mathrm{O}}(hn)$. □

Wegen der trivialen Schranke $\bar{q} \leq n$ ist für $h \in \Omega(n \log n)$ das Kriterium erfüllt. Wegen $h \in \Omega(\log n)$ gilt das gleiche, wenn $\bar{q} \in \mathrm{O}(\sqrt{n})$ sichergestellt wird. Damit ist Satz 6.18 bewiesen. ■

Für sehr unregelmäßige Probleminstanzen führen also selbst vereinzelt auftretende lange Warteschlangen mit hoher Wahrscheinlichkeit nicht zu großen Verzögerungen in der Gesamtausführungszeit. Eine einfache algorithmische Maßnahme zur Verhinderung großer Schlangenlängen besteht darin, bei Überschreiten einer Schranke $\bar{q}$ den normalen Berechnungsablauf zu unterbrechen, und neue Anfragen zu unterbinden, bis alle Warteschlangen geleert sind. Die Auslösung der Unterbrechung läßt sich in Zeit T_{coll} bewerkstelligen. (In Abschnitt A.4.5 wird ein Verfahren beschrieben, das sich für diesen Zweck modifizieren läßt.) Da Anfragen zufällig adressiert werden, ist mit einer Auslösung der Bedingung höchstens nach jeweils $\tilde{\mathrm{O}}(\bar{q}n)$ Anfragen zu rechnen (siehe auch Satz 3.14). Für alle Verbindungsnetzwerke mit $T_{\text{coll}} \in \mathrm{O}(\sqrt{n})$ fällt der Zusatzaufwand für die Unterbrechung also asymptotisch nicht ins Gewicht. Für einigermaßen regelmäßige Probleme mit $h < \bar{q}$ ist außerdem damit zu rechnen, daß die Unterbrechung überhaupt nicht ausgelöst wird, da insgesamt nur $\tilde{\mathrm{O}}(hn)$ Anfragen abgewickelt werden. Wenn diese Annahme[5] gilt, folgt aus Kriterium 6.19 sogar für alle $h \in \mathrm{O}\left(\frac{n}{\log n}\right)$, daß die Warteschlangen nicht zu lang werden.

Eine besonders einfache Maßnahme zur Vermeidung großer Schlangenlängen besteht darin, bei Überschreitung einer maximalen Schlangenlänge alle (lokal) verbleibenden Anfragen abzulehnen. Da damit zu rechnen ist, daß Anfragen schneller abgelehnt werden können als neue Anfragen ankommen, sollten lange Warteschlangen dadurch wirkungsvoll vermieden werden. Da Teilprobleme P, an denen sich lange Warteschlangen gebildet haben, ohnehin schon ein großes $\text{gen}(P)$ angesammelt haben werden, sind die „verlorengegangenen“ Anfragen wohl ebenfalls kein Problem.

6.3 Die Dynamik der Anfangsphase

Es erscheint unökonomisch, daß der Grundalgorithmus für zufälliges Anfragen zu Beginn der Berechnung nur PE 0 mit Arbeit versorgt. Wie bereits angedeutet, werden in Abschnitt 8.4.2 bessere Verfahren untersucht. Trotzdem stellt sich die triviale Initialisierungsstrategie in vielen Fällen als unproblematisch heraus. Hier soll nun anhand eines einfachen Modells untersucht werden, warum dies so ist. In Abschnitt 6.2 konnte der Kommunikationsaufwand nur bis auf konstante Faktoren abgeschätzt werden.

[5] Einem formalen Beweis steht das technische Problem entgegen, daß keine unteren Schranken für den Aufwand einer Anfrage angenommen werden. Deshalb läßt sich theoretisch nicht ausschließen, daß nicht rot gefärbte Anfragen, die in der Analyse bisher nicht berücksichtigt werden mußten, sehr zahlreich sind und die Schlangenlängen beeinflussen.

Um den Preis einiger vereinfachender Annahmen kommt nun die Modellrechnung bis auf kleine additive Konstanten an die ebenfalls durchgeführten Simulationen heran.

Es sei die in Abbildung 6.4 dargestellte synchrone Variante von zufälligem Anfragen mit zufälliger Verschiebung und periodischer Lastverteilung betrachtet. Die aus dem Wurzelproblem erzeugten Teilprobleme seien in der Anfangsphase so groß, daß keine beschäftigten PEs arbeitslos werden. Es wird untersucht, wieviele Iterationen im Mittel benötigt werden, bis alle PEs beschäftigt sind. Die Indikatorzufallsvariable X_{ik} sei genau dann gleich eins, wenn PE i zu Beginn der k-ten Iteration der Hauptschleife Arbeit hat. Sei $U_k = \sum_{i<n} X_{ik}$ die Anzahl beschäftigter PEs zu Beginn von Iteration k. Offenbar gilt $\mathbf{E}U_k = \mathbf{E}\sum_{i<n} X_{ik} = \sum_{i<n} \mathbf{P}[X_{ik} = 1]$. Wird nun nicht PE 0, sondern ein zufälliges PE, mit dem Wurzelproblem initialisiert, so werden alle PEs vollständig gleich behandelt und folglich sind die Zufallsvariablen X_{ik} identisch verteilt. Insbesondere gilt dann $\mathbf{E}U_k = n\mathbf{P}[X_{ik} = 1]$ für ein beliebiges $i < n$. Weiter gilt $\mathbf{P}[X_{i0} = 0] = 1 - 1/n$. Ein PE ist genau dann im Zeitschritt $k+1$ arbeitslos, wenn es im Zeitschritt k bereits arbeitslos war und mit einem PE kommuniziert, das ebenfalls arbeitslos ist. Weiter sei nun angenommen, daß die offensichtlich nutzlosen Verschiebungen um 0 nicht erzeugt werden. Um leichter rechnen zu können, wird außerdem die (formal nicht gerechtfertigte) Annahme gemacht, daß die X_{ik} für festes k unabhängig sind. Da alle Kommunikationspartner gleichwahrscheinlich sind, folgt

$$\mathbf{P}\left[X_{i,k+1} = 0\right] = \mathbf{P}[X_{ik} = 0] \sum_{j \neq i} \frac{1}{n-1} \mathbf{P}\left[X_{jk} = 0\right] = \mathbf{P}[X_{ik} = 0]^2 \quad . \tag{6.2}$$

Nun läßt sich leicht induktiv schließen, daß $\mathbf{P}[X_{ik} = 0] = (1 - 1/n)^{2^k} \leq e^{-2^k/n}$. Für $k = \log n + \log\ln n$ gilt dann $\mathbf{P}[X_i = 0] \leq 1/n$ und damit $\mathbf{E}U_k \geq n(1 - 1/n) = n - 1$. Ist nur noch ein PE arbeitslos, so wird im nächsten Schritt auch dieses beschäftigt. Diese heuristische Betrachtung ist Anlaß für folgende Hypothese:

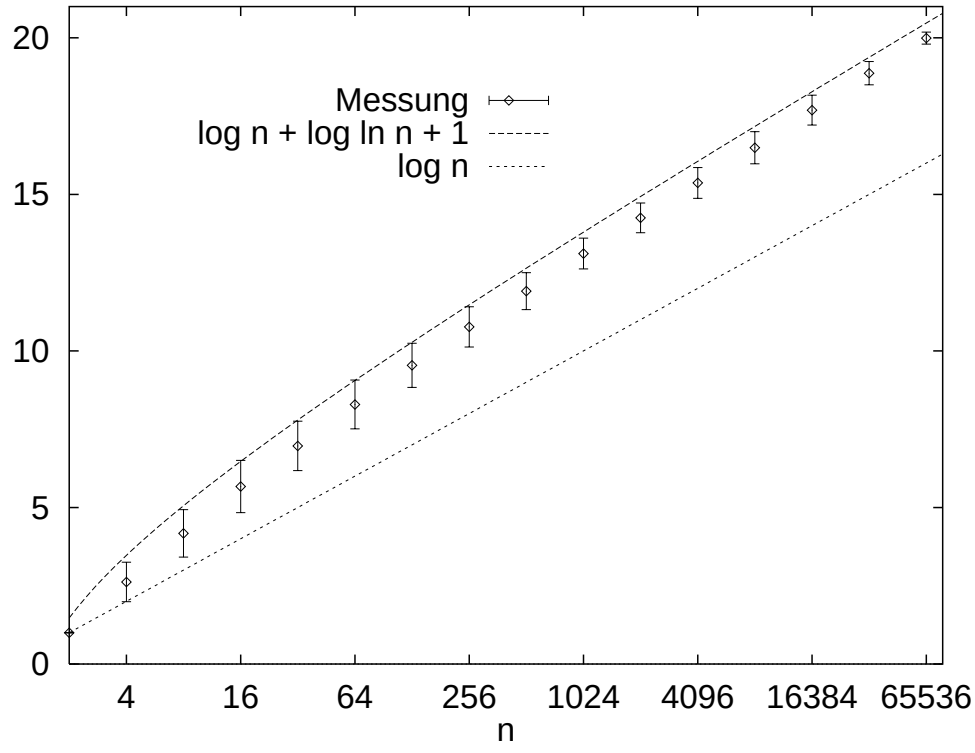

Abbildung 6.7: Anzahl benötigter Iterationen für die Initialisierung.

Hypothese 6.20. *Die mittlere Anzahl Iterationen, die benötigt werden, um allen PEs Arbeit zu verschaffen, läßt sich durch* $\log n + \log\ln n + 1$ *nach oben abschätzen.*

Anstatt zu versuchen, diese Hypothese nun durch langwierige Rechnungen zu einem Satz zu machen, kann sie auch durch Simulation erhärtet werden. Abbildung 6.7 zeigt die mittlere benötigte Anzahl Iterationen für PE-Zahlen n bis zu 2^{16}. Es wurden jeweils 1000 Versuche für alle Zweierpotenzen gemacht. Die Fehlerbalken zeigen die Standardabweichung, die weniger als eine Iteration beträgt. Der gemessene Mittelwert liegt jeweils knapp unter dem von Hypothese 6.20 vorausgesagten Wert. Die gepunktete Linie zeigt $\log n$ – also die Anzahl Iterationen, die ein optimaler deterministischer Initialisierungalgorithmus benötigen würde.

Die Wahrscheinlichkeit, daß die gemessenen Werte nur zufällig unter der vermuteten Schranke liegen, läßt sich außerdem durch den Student t-Test abschätzen. Nach MÜLLER (1991) ergibt sich die Wahrscheinlichkeit

$$1 - A\left(\sqrt{1000}\frac{\bar{T} - (\log n + \log\ln n + 1)}{\sigma} | 999\right)$$

wobei $\bar{T}$ für den gemessenen Mittelwert, σ für die gemessene Standardabweichung und $A(t|\nu)$ für die Verteilungsfunktion der Student t-Verteilung mit ν Freiheitsgraden steht. Im Rahmen der Rechengenauigkeit von Maple V ist dieser Wert für alle gemessenen Fälle 0.

6.4 Zusammenfassung

Zufälliges Anfrage ist ein beweisbar effizientes Lastverteilungsverfahren. Einzig die Tatsache, daß ausschließlich global kommuniziert wird, ist etwas unbefriedigend. Wenn es um konstante Faktoren geht, so gibt es sicher eine Reihe von Verbesserungsmöglichkeiten. So bietet es sich an, zwei Teilprobleme pro PE zu verwalten. Wird ein Teilproblem beendet, so kann auf das andere umgeschaltet werden, während für das andere Ersatz beschafft wird. Dadurch kann die Latenzzeit für die Kommunikation und vor allem die Wartezeit verdeckt werden (insbesondere bei der synchronen Variante). Allerdings ist damit zu rechnen, daß insgesamt mehr Lastübertragungen stattfinden.

Interessant ist auch ein Vergleich mit den in Abschnitt 2.4.2 erklärten Algorithmen, die auf globalem Zählen oder Präfixsummen beruhen und auf einigen Architekturen asymptotisch ebenso effizient sind. (Nicht bei vollständiger Verknüpfung und mehrstufigen Verbindungsnetzwerken.) Um den Preis aufwendigerer Kommunikation werden dort Problemaufspaltungen eingespart, weil Teilprobleme gleichmäßiger aufgespalten werden. Mit Hilfe von Gleichung (3.12) aus Satz 3.14 läßt sich aber leicht zeigen, daß für großes h eine zufällige Zuordnung eine sehr gleichmäßige Verteilung von Anfragen auf Teilprobleme garantiert. Gerade bei schwierigen, unregelmäßigen Problemen ist zufälliges Anfragen also vorzuziehen.

Ergebnisse

Diese Arbeit enthält die erste[6)] Analyse von zufälligem Anfragen, die asymptotisch genaue Laufzeitaussagen macht. Dabei ist zu beachten, daß es sich nicht um irgendeinen Algorithmus handelt, sondern um den wahrscheinlich besten gegenwärtig praktisch eingesetzten Algorithmus für parallele Tiefensuche. Die Variante mit zufälligem Verschieben und das damit

[6)] BLUMOFE UND LEISERSON (1994) analysiert ein verwandtes Modell. Die hier vorgestellte Analyse wurde in vereinfachter Form aber bereits am 28. April 1994 veröffentlicht (SANDERS, 1994a). Der Preprint von BLUMOFE UND LEISERSON (1994) (den ich Ende 94 erhalten habe) trägt das Datum 3. Mai 94!

verbundene Ergebnis zum Verbrauch von Zufallsbits ist neu. Die Dynamik der Anfangsphase wurde bisher nicht genau untersucht. Insbesondere beim asynchronen Algorithmus wird ein deutlich allgemeineres und praxisnäheres Modell verwendet als dies üblich ist. Gerade die Behandlung von Totzeiten und Warteschlangen fällt in anderen Arbeiten weitgehend unter den Tisch. Zum Beispiel werden in BLUMOFE UND LEISERSON (1994) zwar Warteschlangen modelliert, aber die Lastverteilung erfolgt in synchronen Phasen in denen jeweils eine Anfrage pro PE ausgeliefert wird und eine Antwort gegeben wird. Dadurch vereinfacht sich die Analyse erheblich; weil Totzeiten ausgeschlossen sind; weil klar ist, welche Anfragen welche anderen beeinflussen können; und weil sich zeigen läßt, daß Warteschlangen dann sehr kurz bleiben. Die Berücksichtigung der Maschinen- und Problemparameter d, b, h, l, T_{split} und T_{atomic}, auf die in anderen Arbeiten weitgehend verzichtet wird, ist zwar meist nicht sehr schwierig, aber doch wichtig für eine genaue Interpretation der Ergebnisse.

Kapitel 7

Fragen-und-Mischen

Der in Kapitel 6 untersuchte Algorithmus für zufälliges Anfragen ist bezogen auf die Anzahl benötigter Kommunikation bereits asymptotisch optimal (zumindest unter allen empfängerveranlaßten Algorithmen). Die Kommunikationen sind allerdings global, und es stellt sich die Frage, ob sich dies nicht verbessern läßt. Leider sind alle bekannten Algorithmen, die ausschließlich lokal kommunizieren, schlechter, weil Last dadurch „verklumpt“ (siehe auch Abschnitt 2.4.2). Diesem Kapitel liegt die Idee zugrunde, globale und lokale Kommunikation geeignet zu mischen, um so die Vorteile beider Verfahren zu vereinigen. In Abschnitt 7.1 wird ein einfacher synchroner Algorithmus für Hyperwürfel eingeführt, der sich als asymptotisch optimaler empfängerveranlaßter Algorithmus herausstellt. Nachbarschaftskommunikation wird dort mit gelegentlichem zufälligem Vertauschen von Teilproblemen kombiniert. Dieser Algorithmus wird in Abschnitt 7.2 für andere Verbindungsnetzwerke angepaßt. Abschnitt 7.3 zeigt, wie globale Synchronisationen vermieden werden können. In Abschnitt 7.4 wird gezeigt, wie Vertauschungen der Teilprobleme eingespart werden können. Weitere Verfeinerungen und vor allem eine Kombination der Vorzüge von Fragen-und-Mischen und zufälligem Anfragen sind Gegenstand von Abschnitt 7.5. Abschnitt 7.6 faßt das Kapitel zusammen.

7.1 Ein synchroner Hyperwürfelalgorithmus

7.1.1 Der Algorithmus

Ausgangspunkt für dieses Kapitel ist der in Abbildung 7.1 dargestellte Algorithmus. Um die Analyse zu vereinfachen, ist er bewußt einfach gehalten. In der Tat ähnelt er stark dem synchronen Algorithmus mit periodischer Lastverteilung für zufälliges Anfragen aus Abbildung 6.4. Es wird aber nicht global kommuniziert, sondern mit einem Nachbarn im Hyperwürfel (entsprechend muß n eine Zweierpotenz sein).

Mit welchem Nachbarn kommuniziert wird, legt der Dimensionszähler i fest, der in einer synchronen Schleife zyklisch durchlaufen wird. Die Berechnung läßt sich in *Zyklen* für jedes Durchlaufen der äußeren Schleife einteilen. Jeder Zyklus ist wiederum in $\log n$ Phasen mit den Nummern $i \in \mathbb{N}_n$ eingeteilt. Wie in Abschnitt 2.4.2 bereits erklärt, würde eine Be-

```
var P : Subproblem                                                    1
P := if i_PE = 0 then P_root else P_∅                                 2
while not finished do synchronously                                   3
    for i:=0 to log n − 1 do synchronously                            4
        P := work(P, Δt)                    (* Do some sequential work *) 5
        if T(P) = 0 then                                              6
            (P, P⟨i_PE ⊕ 2^i⟩) := split(P⟨i_PE ⊕ 2^i⟩)                 7
        fi                                                            8
    endfor                                                            9
    globally permute subproblems randomly                             10
```

Abbildung 7.1: Grundalgorithmus für Fragen-und-Mischen.

schränkung auf Nachbarschaftskommunikation zu einer Verklumpung der Last führen. Um dies zu verhindern, werden sämtliche Teilprobleme nach jedem Zyklus zufällig permutiert. Hauptaufgabe der Analyse wird es sein zu zeigen, daß dies ausreicht um Verklumpungen zu verhindern. Abbildung 7.2 zeigt die Einteilung der Zeitachse für $n = 2^3$ und 2 Zyklen.

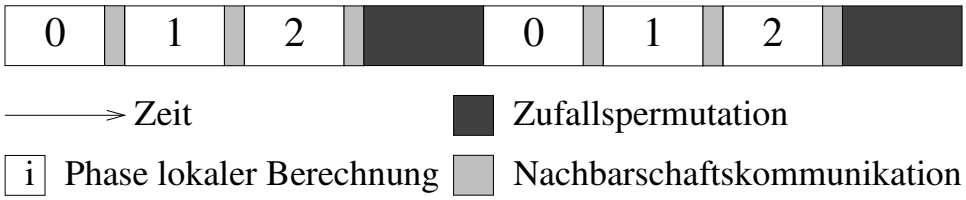

Abbildung 7.2: Zeiteinteilung für Fragen-und-Mischen

7.1.2 Analyse

Die Analyse wird gleich für Hyperwürfel mit eingeschränkten Fähigkeiten durchgeführt. Es wird nämlich gezeigt, daß Algorithmus 7.1 sich effizient als *normaler* Hyperwürfelalgorithmus im Sinne von Abschnitt 2.2.1 implementieren läßt. Die vollen Fähigkeiten eines Hyperwürfels werden nicht wirklich benötigt, und die Übertragung des Ergebnisses auf hyperwürfelartige Netze in Abschnitt 7.2.1 wird dadurch einfach.

Satz 7.1. *Es gibt für alle Konstanten* $\varepsilon > 0$ *eine Wahl von* Δt*, so daß für die Ausführungszeit von Algorithmus 7.1 als normaler Hyperwürfelalgorithmus gilt*

$$T_{\text{par}} \preceq \left(1 + \varepsilon + \varepsilon\tilde{\mathrm{O}}\left(\frac{1}{\log\log n}\right)\right)\frac{T_{\text{seq}}}{n} + \tilde{\mathrm{O}}\left(T_{\text{atomic}} + h(l + T_{\text{split}})\right) \quad .$$

Da die Nachbarschaftskommunikation mit ihrer strengen Dimensionenabfolge bereits einem normalen Algorithmus entspricht, ist nur noch darauf zu achten, daß die Zufallspermutation ebenfalls durch einen normalen Algorithmus realisiert wird. Die Aussage ist aus technischen Gründen etwas komplizierter, als die für zufälliges Anfragen. Der zur sequentiellen

Ausführungszeit proportionale Term der parallelen Ausführungszeit ist eine Zufallsvariable. Diese Schwankung ist aber mit hoher Wahrscheinlichkeit klein; vor allem für große n.

Die Grundstruktur der Analyse ähnelt der Analyse von zufälligem Anfragen. Die Hauptschwierigkeit besteht darin, zu zeigen, daß Nachbarschaftskommunikation plus gelegentliche Zufallspermutationen sich ähnlich verhält wie die globale Kommunikation von zufälligem Anfragen. Den Schlüssel dazu liefert Lemma 7.4.

Definition 7.2. Ein i-Würfel bezeichne einen Teilgraphen eines Hyperwürfels, mit den PE-Nummern $k2^i \leq j < (k+1)2^i$ für $k \in \mathbb{N}_{n/2^i}$.

Werden in einem Hyperwürfel alle Kanten mit Dimensionsnummer $\geq i$ weggelassen, so zerfällt er in $2^{\log n - i}$ i-dimensionale Hyperwürfel. Dies sind gerade die in ihm enthaltenen i-Würfel.

Definition 7.3. Seien S, T beliebige Teilprobleme. Dann bezeichne $S \overset{i}{—} T$ das Ereignis[1)], daß S und T auf PEs liegen, die entlang Dimension i benachbart sind.

Wenn s und t also die Nummern der PEs sind auf denen S und T liegen, so gilt $s = t \oplus 2^i$. Vereinfacht ausgedrückt, geht es nun darum, zu zeigen, daß in jeder Iteration (fast) alle Paarungen der Form $S \overset{i}{—} T$ gleichwahrscheinlich sind. Dies gilt allerdings nicht für Teilprobleme, die im gleichen i-Würfel liegen. Außerdem werden i.allg. nicht alle Permutationen zur gleichen Menge vorhandener Teilprobleme nach Phase i führen. Deshalb wird eine Fallunterscheidung gemacht, indem die Menge der Zufallspermutationen in Äquivalenzklassen eingeteilt wird, innerhalb derer der Systemzustand sich nur durch die relative Lage der Teilprobleme unterscheidet.

Lemma 7.4. *Betrachtet sei ein beliebiger Zyklus von Algorithmus 7.1 zu einem Zeitpunkt, an dem entlang Dimension i kommuniziert wird. Sei S ein beliebiges Teilproblem. Sei $\mathcal{T}$ die Menge aller Teilprobleme, die* nicht *im gleichen i-Würfel liegen wie S. Leere Teilprobleme seien unterscheidbar. Dann gilt für beliebige $T, T' \in \mathcal{T}$, daß* $\mathbf{P}\left[S \overset{i}{—} T \mid (S, \mathcal{T})\right] = \mathbf{P}\left[S \overset{i}{—} T' \mid (S, \mathcal{T})\right]$. *$(S, \mathcal{T})$ bezeichne dabei die Teilmenge des Ereignisraums, in der zum betrachteten Zeitpunkt das gleiche Teilproblem S und die gleiche Teilproblemmenge $\mathcal{T}$ hervorgebracht wird.*

Beweis. Es sei zunächst davon ausgegangen, daß „work“ und „split“ deterministisch arbeiten, also selbst keinen Zufallsgenerator verwenden. Sei M die Untergruppe der Permutationen (S_n), die durch Austausch ganzer i-Würfel und Spiegelungen von i-Würfeln erzeugt werden. Formal ausgedrückt sind die Vertauschungen $V := \left\{ v_{mm'} \mid m, m' \in \mathbb{N}_{n/2^i} \right\}$ durch

$$v_{mm'}(k) := \begin{cases} k + 2^i(m' - m) & \text{falls } m2^i \leq k < (m+1)2^i \\ k + 2^i(m - m') & \text{falls } m'2^i \leq k < (m'+1)2^i \\ k & \text{sonst} \end{cases}$$

[1)] Ein *Ereignis* bezeichnet in der Stochastik eine Menge von Zuständen (MÜLLER, 1991).

und die Spiegelungen $G := \left\{ s_{mj} \,|\, m \in \mathbb{N}_{n/2^i}, j \in \mathbb{N}_i \right\}$ durch

$$s_{mj}(k) := \begin{cases} k \oplus 2^j & \text{falls } m2^i \leq k < (m+1)2^i \\ k & \text{sonst} \end{cases}$$

definiert. Dann ist $M :=< V \cup G >$ das Erzeugnis von $V \cup G$. Aus der elementaren Gruppentheorie ist bekannt, daß M (wie jede Untergruppe) benutzt werden kann, um S_n durch Nebenklassenbildung in Äquivalenzklassen zu partitionieren. Für jedes $\pi \in S_n$ kann eine Äquivalenzklasse $C_\pi := \pi M = \{\pi m \,|\, m \in M\}$ definiert werden.

Sei nun π die Permutation, die zur betrachteten Situation geführt hat. Es genügt zu zeigen, daß

$$\left| C_T := \left\{ \sigma \in C_\pi \,|\, S \xrightarrow{i} T \right\} \right| = \left| C_{T'} := \left\{ \sigma \in C_\pi \,|\, S \xrightarrow{i} T' \right\} \right| \quad .$$

Obwohl dies auch direkt durch Zählen der Kardinalität der beiden Mengen möglich ist, ist es einfacher und eleganter, zu zeigen, daß eine Bijektion $f_{TT'} \in {C_{T'}}^{C_T}$ existiert. Aus Symmetriegründen genügt es sogar, eine Injektion anzugeben. Die folgende Vorschrift leistet das Gewünschte:

- Spiegle den i-Würfel, in dem sich T befindet, solange, bis T innerhalb seines i-Würfels die gleiche Position hat wie T'. Für jede Bitposition kleiner i, an der sich die PE-Nummern der Teilprobleme unterscheiden, wird eine Spiegelung durchgeführt.
- Vertausche die i-Würfel in denen T und T' liegen.

Formal läßt sich $f_{TT'}$ folgendermaßen definieren: Seien $t2^i + u$ bzw. $t'2^i + u'$ die Nummern der PEs auf denen T bzw. T' liegen mit $0 \leq u, u' < 2^i$. Sei B die Menge aller Bitpositionen an denen sich die Binärrepräsentationen von u und u' unterscheiden. Die Operation $\circ$ stehe für die Hintereinanderausführung von Permutationen.

$$f_{TT'} := \left(\bigcirc_{j \in B} s_{tj} \right) \circ v_{tt'} \quad .$$

$f_{TT'}$ bewegt sich per Definition in C_π und $\forall \sigma \in C_T : \sigma \circ f_{TT'} \in C_{T'}$. Da sowohl i-Würfelvertauschungen (Elemente von V) als auch i-Würfelspiegelungen (Elemente von G) injektiv sind und weil Verknüpfungen injektiver Funktionen wieder injektiv sind, muß auch $f_{TT'}$ injektiv sein. Damit ist gezeigt, daß $|C_T| = |C_{T'}|$, und es folgt die Behauptung.

Benutzen schließlich „work“ und „split“ selbst Zufallsgeneratoren, so kann eine Fallunterscheidung für jeden möglichen Ausgang der dort getroffenen Zufallsentscheidungen gemacht werden. In jedem einzelnen Fall ist der obige Beweis anwendbar. Folglich gilt die Aussage auch insgesamt. □

Nun kann gezeigt werden, daß in den meisten Phasen mit schlechter PE-Auslastung jedes Teilproblem mit einer gewissen Wahrscheinlichkeit aufgespalten wird.

Lemma 7.5. *Für beliebiges aber festes $\gamma \in (0,1)$, für ein beliebiges Teilproblem S und für eine beliebige Phase mit Nummer $0 \leq i < \log n - \log \frac{2}{\gamma}$ gilt: wenn irgendwann während der Phase mindestens γn PEs arbeitslos sind, so wird S nach dieser Phase mit einer Mindestwahrscheinlichkeit $\gamma/2$ aufgespalten.*

Beweis. Da die Anzahl beschäftigter PEs während einer Phase nur abnehmen kann, gibt es nach der Phase mindestens γn arbeitslose PEs. Im i-Würfel in dem sich S befindet, gibt es aber nur $2^i \leq n\gamma/2$ Teilprobleme. Folglich gibt es mindestens $n\gamma/2$ leere Teilprobleme, die sich außerhalb des i-Würfels von S befinden. Nach Lemma 7.4 ist jedes Teilproblem außerhalb dieses i-Würfels mit gleicher Wahrscheinlichkeit das Nachbarteilproblem mit dem in Phase i Last ausgetauscht wird. Die Wahrscheinlichkeit, daß es sich dabei um ein leeres Teilproblem handelt ist also mindestens

$$\frac{n\gamma/2}{n-2^i} \geq \gamma/2 \ .$$

□

Jetzt ist es nicht mehr schwer zu zeigen, daß es nur wenige Iterationen mit schlechter PE-Auslastung geben kann.

Lemma 7.6. *Es genügen* $\tilde{\mathrm{O}}(h)$ *Iterationen von Algorithmus 7.1 an deren Ende mindestens* γn *PEs ein leeres Teilproblem haben und die eine Phasennummer kleiner* $\log n - \log \frac{2}{\gamma}$ *haben, damit es kein Teilproblem* P *mit* $\mathrm{gen}(P) < h$ *mehr gibt.*

Beweis. Der Beweis von Lemma 6.2 kann weitgehend übernommen werden. „Rote“ Iterationen sind nun Phasen mit den in den Voraussetzungen spezifizierten Eigenschaften. In jeder dieser Phasen werden genau $\gamma/2$ arbeitslose PEs außerhalb des i-Würfels von $P\langle i\rangle$ rot markiert. Anstelle der Wahrscheinlichkeit $1-e^{-m/n}$ tritt $\gamma/2$. Die einzelnen Phasen sind unabhängig, weil nach Lemma 7.4 die Wahrscheinlichkeit, in einer roten Phase ein rotes Teilproblem zum Nachbarn zu haben, immer gleich ist und nicht von anderen Phasen abhängt. □

Völlig analog zu Lemma 6.4 kann auch die Anzahl Iterationen mit hoher PE-Auslastung begrenzt werden:

Lemma 7.7. *Sei* $\gamma \in [0,1)$ *beliebig. Nach höchstens* $\left\lceil \frac{T_{\mathrm{seq}}}{n(1-\gamma)\Delta t} \right\rceil$ *Phasen von Algorithmus 7.1, die an ihrem jeweiligen Ende höchstens* γn *PEs mit leerem Teilproblem haben, gibt es keine nichtleeren Teilprobleme mehr.*

Ähnlich zum Beweis von Satz 6.1 gilt es nun, Werte für Δt und γ zu finden, so daß die behaupteten Schranken für die Ausführungszeit gelten. Da es Zyklen und Phasen zu unterscheiden gilt, sind aber zunächst noch einige weiterer Hilfssätze zwecks modularer Behandlung des Beweises angebracht.

Lemma 7.8. *Die Ausführungszeit für einen Durchlauf der* **for***-Schleife von Algorithmus 7.1 (eine Phase) ist in* $T_{\mathrm{phase}} := \Delta t + T_{\mathrm{split}} + \mathrm{O}(l)$.

Beweis. Der Aufwand für die sequentielle Berechnung ist Δt. Der Kontrollaufwand ist konstant. Für das Aufspalten von Problemen fällt ein Aufwand von T_{split} an. Da nur Nachbarschaftskommunikation stattfindet, ist der Kommunikationsaufwand in $\mathrm{O}(l)$. Man beachte insbesondere, daß *nicht* nach jeder Phase eine Terminierungserkennung durchgeführt wird. □

Lemma 7.9. *Die Ausführungszeit* T_{cycle} *für einen Durchlauf der* **while***-Schleife von Algorithmus 7.1 (einen Zyklus) ist in*

$$\log n \left(\Delta t + T_{\mathrm{split}} + \mathrm{O}(l) + \tilde{\mathrm{O}}\left(\frac{l}{\log\log n} \right) \right) \ .$$

Beweis. Der Aufwand für $\log n$ Iterationen ist nach Lemma 7.8 in $\log n(\Delta t + T_{\text{split}} + \mathrm{O}(l))$. Die Berechnung einer Zufallspermutation mit einem normalen Hyperwürfelalgorithmus benötigt eine Zeit in $\mathrm{O}(\log n) + \tilde{\mathrm{O}}\left(\frac{\log n}{\log\log n}\right)$ wenn Algorithmus 4.2 verwendet wird. Das Vertauschen der Teilprobleme entsprechend der so berechneten Permutation ist in Zeit $\mathrm{O}(l\log n) + \tilde{\mathrm{O}}\left(\frac{l\log n}{\log\log n}\right)$ möglich.[2)] Schließlich kann eine Terminierungserkennung durchgeführt werden, indem in Zeit $\mathrm{O}(\log n)$ eine globale Und-Reduktion über das Prädikat $T(P) = 0$ durchgeführt wird. □

Der Beweis von Satz 7.1 läßt sich nun durch eine einfache wenn auch etwas umfangreiche Betrachtung zu Ende bringen: Über Phasen mit Nummer $i \geq \log\frac{2}{\gamma}$ kann wenig gesagt werden. Von den verbleibenden gibt es aber nach Lemma 7.7 höchstens $\left\lceil \frac{T_{\text{seq}}}{n(1-\gamma)\Delta t} \right\rceil$ viele mit hoher PE-Auslastung. Nach Lemma 7.7 genügen $\tilde{\mathrm{O}}(h)$ Phasen mit niedriger Auslastung, damit es nur noch atomare Teilprobleme gibt. Nach weiteren $\left\lceil \frac{T_{\text{atomic}}}{\Delta t} \right\rceil$ Phasen müssen auch die verbleibenden atomaren Teilprobleme abgearbeitet sein. Da es in jedem Zyklus mindestens

$$\left\lfloor \log n - \log\frac{2}{\gamma} \right\rfloor > \log n - \log\frac{2}{\gamma} - 1 = \log n - \log\frac{4}{\gamma}$$

Phasen mit niedriger Nummer gibt, genügen (für $n > \frac{4}{\gamma}$

$$\left\lceil \frac{\left\lceil \frac{T_{\text{seq}}}{n(1-\gamma)\Delta t} \right\rceil + \tilde{\mathrm{O}}(h) + \left\lceil \frac{T_{\text{atomic}}}{\Delta t} \right\rceil}{\log n - \log\frac{4}{\gamma}} \right\rceil \preceq \frac{\frac{T_{\text{seq}}}{n(1-\gamma)\Delta t} + \tilde{\mathrm{O}}(h) + 2 + \frac{T_{\text{atomic}}}{\Delta t} + \log n}{\log n - \log\frac{4}{\gamma}}$$
$$\preceq \frac{\frac{T_{\text{seq}}}{n(1-\gamma)\Delta t} + \tilde{\mathrm{O}}(h) + \frac{T_{\text{atomic}}}{\Delta t}}{\log n - \log\frac{4}{\gamma}}$$

Zyklen für die Lösung der gesamten Probleminstanz. (Man beachte, daß $h \in \Omega(\log n)$ angenommen wird.) Nach Lemma 7.9 gibt es eine Konstante c, so daß

$$T_{\text{cycle}} \leq \log n \left(\Delta t + T_{\text{split}} + cl + \tilde{\mathrm{O}}\left(\frac{l}{\log\log n}\right)\right) \quad .$$

Damit ergibt sich eine Schranke für die parallele Ausführungszeit.

$$\begin{aligned} T_{\text{par}} &\preceq \frac{\frac{T_{\text{seq}}}{n(1-\gamma)\Delta t} + \tilde{\mathrm{O}}(h) + \frac{T_{\text{atomic}}}{\Delta t}}{\log n - \log\frac{4}{\gamma}} \log n \left(\Delta t + T_{\text{split}} + cl + \tilde{\mathrm{O}}\left(\frac{l}{\log\log n}\right)\right) \\ &= \frac{\log n}{\log n - \log\frac{4}{\gamma}} \left(\frac{T_{\text{seq}}}{n(1-\gamma)\Delta t} + \tilde{\mathrm{O}}(h) + \frac{T_{\text{atomic}}}{\Delta t}\right) \left(\Delta t + T_{\text{split}} + cl + \tilde{\mathrm{O}}\left(\frac{l}{\log\log n}\right)\right) \quad . \end{aligned} \tag{7.1}$$

[2)] Auf Maschinen, die in allen Dimensionen gleichzeitig kommunizieren können, ist das sogar in Zeit $\tilde{\mathrm{O}}(\log n + l)$ möglich. Die einzige dadurch bewirkte asymptotische Änderung im Gesamtergebnis ist aber, daß das $\tilde{\mathrm{O}}(1/\log\log n)$ aus Satz 7.1 durch einen etwas schneller gegen Null konvergierenden Term ersetzt werden kann.

Der Faktor $\frac{\log n}{\log n - \log \frac{4}{\gamma}}$ liegt für hinreichend große n beliebig nahe bei 1. Zu gegebenem ε sei im folgenden n hinreichend groß gewählt, damit

$$\frac{\log n}{\log n - \log \frac{4}{\gamma}} \leq \sqrt{1+\frac{\varepsilon}{2}} \ .$$

T_{par} läßt sich als Summe $T_1 + T_2 + T_3$ darstellen, wobei T_1 für die von T_{seq} abhängigen Terme, T_2 für die von h abhängigen Terme und T_3 für die von T_{atomic} abhängigen Terme steht. Diese Summanden seien nun getrennt betrachtet.

Lemma 7.10. *Wird* $\gamma = 1 - 1/\sqrt{1+\frac{\varepsilon}{2}}$ *und* $\Delta t = 2(1+\frac{1}{\varepsilon})(T_{\text{split}} + cl)$ *gewählt, so gilt*

$$T_1 \preceq \frac{T_{\text{seq}}}{n}\left(1+\varepsilon+\varepsilon\tilde{\mathrm{O}}\left(\frac{1}{\log\log n}\right)\right) \ .$$

Beweis. Es gilt $\sqrt{1+\frac{\varepsilon}{2}}\frac{1}{(1-\gamma)} = 1+\frac{\varepsilon}{2}$. Damit ergibt sich

$$\begin{aligned}
T_1 \preceq & \frac{T_{\text{seq}}}{n}\left(1+\frac{\varepsilon}{2}\right)\left(1+\frac{\tilde{\mathrm{O}}\left(\frac{l}{\log\log n}\right)+T_{\text{split}}+cl}{2(1+\frac{1}{\varepsilon})(T_{\text{split}}+cl)}\right) \\
= & \frac{T_{\text{seq}}}{n}\left(1+\frac{\varepsilon}{2}+\frac{1}{2}(1+\frac{\varepsilon}{2})\frac{1}{1+\frac{1}{\varepsilon}}\left(\frac{\tilde{\mathrm{O}}\left(\frac{l}{\log\log n}\right)}{T_{\text{split}}+cl}+1\right)\right) \\
= & \frac{T_{\text{seq}}}{n}\left(1+\frac{\varepsilon}{2}+\frac{\varepsilon}{2}\frac{2+\varepsilon}{2+2\varepsilon}\left(\frac{\tilde{\mathrm{O}}\left(\frac{l}{\log\log n}\right)}{T_{\text{split}}+cl}+1\right)\right) \\
\preceq & \frac{T_{\text{seq}}}{n}\left(1+\frac{\varepsilon}{2}+\frac{\varepsilon}{2}\left(\tilde{\mathrm{O}}\left(\frac{1}{\log\log n}\right)+1\right)\right) \\
= & \frac{T_{\text{seq}}}{n}\left(1+\varepsilon+\varepsilon\tilde{\mathrm{O}}\left(\frac{1}{\log\log n}\right)\right)
\end{aligned}$$

□

Lemma 7.11. $T_2 \in \tilde{\mathrm{O}}\left(h(T_{\text{split}}+l)\right)$.

Beweis.

$$\begin{aligned}
T_2 \preceq & \sqrt{1+\frac{\varepsilon}{2}}\tilde{\mathrm{O}}(h)\left(\Delta t + T_{\text{split}} + cl + \tilde{\mathrm{O}}\left(\frac{l}{\log\log n}\right)\right) \\
= & \mathrm{O}(1)\,\tilde{\mathrm{O}}(h)\left(2(1+\frac{1}{\varepsilon})(T_{\text{split}}+cl) + T_{\text{split}} + cl + \tilde{\mathrm{O}}\left(\frac{l}{\log\log n}\right)\right) \\
= & \tilde{\mathrm{O}}\left(h(T_{\text{split}}+l)\right)
\end{aligned}$$

□

Lemma 7.12. $T_3 \in \tilde{\mathrm{O}}(T_{\text{atomic}})$.

Beweis.

$$\begin{aligned} T_3 &\preceq \sqrt{1+\frac{\varepsilon}{2}}\frac{T_{\text{atomic}}}{\Delta t}\left(\Delta t + T_{\text{split}} + cl + \tilde{\mathrm{O}}\left(\frac{l}{\log\log n}\right)\right) \\ &\preceq \mathrm{O}(1)\left(T_{\text{atomic}} + \frac{T_{\text{atomic}}}{2(1+\frac{1}{\varepsilon})} + \tilde{\mathrm{O}}(1)\right) = \tilde{\mathrm{O}}(T_{\text{atomic}}) \end{aligned}$$

□

Insgesamt folgt aus den Lemmata 7.10, 7.11 und 7.12

$$\begin{aligned} T_{\text{par}} &\preceq T_1 + T_2 + T_3 \\ &\preceq \frac{T_{\text{seq}}}{n}\left(1+\varepsilon+\varepsilon\tilde{\mathrm{O}}\left(\frac{1}{\log\log n}\right)\right) + \tilde{\mathrm{O}}\left(h(T_{\text{split}}+l)\right) + \tilde{\mathrm{O}}(T_{\text{atomic}}) \\ &= \frac{T_{\text{seq}}}{n}\left(1+\varepsilon+\varepsilon\tilde{\mathrm{O}}\left(\frac{1}{\log\log n}\right)\right) + \tilde{\mathrm{O}}\left(T_{\text{atomic}} + h(l+T_{\text{split}})\right) \end{aligned}$$

■

Fragen-und-Mischen benötigt asymptotisch nicht mehr Teilproblemübertragungen als zufälliges Anfragen. Außerdem fallen die globalen Kommunikationen gegenüber der Nachbarschaftskommunikation nicht ins Gewicht. Damit ist der Algorithmus optimal unter allen empfängerveranlaßten Algorithmen und kann von keinem deterministischen Algorithmus übertroffen werden. Für $l \notin \Omega(\log n)$ ist Fragen-und-Mischen schneller als zufälliges Anfragen. Der Kommunikationsaufwand ist um einen Faktor $\Theta\left(\max(\frac{\log n}{l}, 1)\right)$ geringer. Für normale Hyperwürfelalgorithmen hat der Algorithmus für alle l einen um einen Faktor $\log n$ geringeren Kommunikationsaufwand als zufälliges Anfragen.

7.2 Andere Netzwerke

Der Fragen-und-Mischen-Algorithmus läßt sich unmittelbar auf andere Verbindungsnetzwerke anwenden, indem ein virtueller Hyperwürfel auf der physikalischen Architektur emuliert wird. Dabei kann jedoch aus einer lokalen Hyperwürfelkommunikation eine nichtlokale Kommunikation werden. Hier wird untersucht, inwieweit dieser Ansatz dennoch zu einem asymptotisch effizienteren Algorithmus als zufälliges Anfragen führt. In Abschnitt 7.2.1 zeigt sich wie angedeutet, daß dies auf zum Hyperwürfel verwandten Netzen sehr einfach ist. Abschnitt 7.2.2 modifiziert den Algorithmus so, daß auch auf Gittern auf globale Kommunikation verzichtet werden kann. Schwieriger ist die Situation auf vielen mehrstufigen Verbindungsnetzwerken, da dort der Begriff lokale Kommunikation keinen Sinn macht. Immerhin kann in Abschnitt 7.2.3 gezeigt werden, daß auf Fat trees eine asymptotische Verringerung der Nachrichtenlatenz möglich ist.

7.2.1 Hyperwürfelartige Netze

Die bereits in Abschnitt 2.2.1 eingeführten hyperwürfelartigen Verbindungsnetze zeichnen sich nach LEIGHTON (1992) dadurch aus, daß sie normale Hyperwürfelalgorithmen unter einem konstanten Faktor Zeitverlust emulieren können.

Satz 7.13. *Es gibt für alle Konstanten* $\varepsilon > 0$ *eine Wahl von* Δt*, so daß für die Ausführungszeit von Algorithmus 7.1 auf* hyperwürfelartigen Netzen *gilt*

$$T_{\text{par}} \preceq \left(1+\varepsilon+\varepsilon\tilde{\mathrm{O}}\left(\frac{1}{\log\log n}\right)\right)\frac{T_{\text{seq}}}{n}+\tilde{\mathrm{O}}\left(T_{\text{atomic}}+h(l+T_{\text{split}})\right) \quad .$$

Beweis. Da Satz 7.1 das Gewünschte bereits für normale Hyperwürfelalgorithmen aussagt, bleibt zu zeigen, daß nur die Kommunikation durch die Hyperwürfelemulation um einen konstanten Faktor verlangsamt wird, nicht aber die sequentiellen Berechnungsphasen. Dies ist nicht selbstverständlich, da die Emulationen in LEIGHTON (1992) so aufgebaut sind, daß die Berechnungen von Schritt zu Schritt migrieren. Glücklicherweise wird für jede Kommunikationsphase ohnehin eine komplette Teilproblemübertragung einkalkuliert. □

Auch für hyperwürfelartige Netze ist Fragen-und-mischen im gleichen Sinne asymptotisch optimal wie für Hyperwürfel. Es ist interessant, daß die höhere Bandbreite von echten Hyperwürfeln keinen asymptotischen Vorteil bringt.

7.2.2 Gitter

Hyperwürfel lassen sich auf recht naheliegende Weise in Gitter einbetten. Einige Varianten sind in KUMAR ET AL. (1994) und GMEHLICH (1994) beschrieben. Es sei zunächst von einem würfelförmigen r-dimensionalen Gitter mit einer Zweierpotenz als Seitenlänge ausgegangen. Hier wird die folgende Einbettung verwendet: Sei $N = \log n$. Sei $(b_0,\ldots,b_{N-1})$ die Binärdarstellung einer PE-Nummer des Hyperwürfels. Die k-te Koordinate ($k \in \mathbb{N}_r$) der Gitter-PEs, auf das dieses Hyperwürfel-PE abgebildet wird, hat die Binärdarstellung $(b_k, b_{k+r}, \ldots, b_{N-r+k})$. Abbildung 7.3 zeigt die Einbettung für $r = 2$ und $n = 16$.

Wird diese Einbettung für Algorithmus 7.1 verwendet, so ergibt sich ein Algorithmus mit einem um einen Faktor $\mathrm{O}(\log n)$ geringeren Kommunikationsaufwand als zufälliges Anfragen. Dies ist für $h \in \mathrm{O}(\log n)$ und $l \in \mathrm{O}(1)$ asymptotisch optimal. Allgemeinere Optimalitätsaussagen scheitern daran, daß die unteren Schranken aus Kapitel 5 nicht ausreichend zwischen globaler und lokaler Kommunikation differenzieren.

Satz 7.14. *Es gibt für alle Konstanten* $\varepsilon > 0$ *eine Wahl von* $\Delta t \in \Theta\left(T_{\text{split}} + l\frac{n^{1/r}}{\log n}\right)$*, so daß für die Ausführungszeit von Algorithmus 7.1 auf Gittern gilt*

$$T_{\text{par}} \preceq (1+\varepsilon)\frac{T_{\text{seq}}}{n}+\tilde{\mathrm{O}}\left(T_{\text{atomic}}+h(\frac{ln^{1/r}}{\log n})+T_{\text{split}}\right) \quad .$$

Es fällt auf, daß der von T_{seq} abhängige Term von T_{par} nun keine Zufallsvariable mehr ist. Bei Gittern sind nämlich effiziente deterministische Algorithmen für Datentransport und zur Bestimmung von Zufallspermutationen bekannt. Dadurch läßt sich eine deterministische Schranke für die Ausführungszeit eines Zyklus angeben.

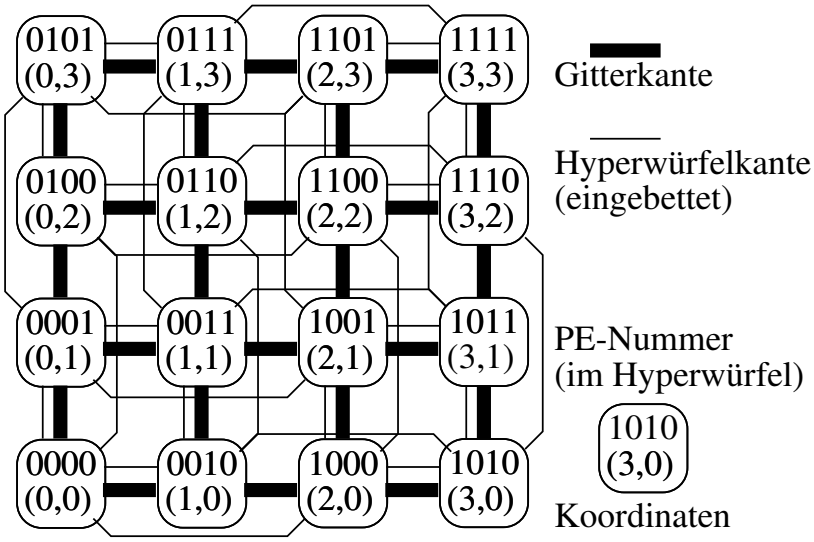

Abbildung 7.3: Einbettung eines 4-D Hyperwürfels in ein 4×4-Gitter

Lemma 7.15. $T_{\text{cycle}} \preceq \log n(T_{\text{split}} + \Delta t) + \mathrm{O}\left(ln^{1/r}\right)$.

Beweis. Der Aufwand für sequentielles Bearbeiten, Spalten und Kontrollaufwand ist in $\log n(\Delta t + T_{\text{split}} + \mathrm{O}(1))$. Eine Zufallspermutation kann in Zeit $\mathrm{O}\left(n^{1/r}\right)$ bestimmt werden, wenn einer der in HAGERUP (1991) erwähnten Algorithmen mit deterministischer Ausführungszeit verwendet wird.[3)] Das Durchführen der Permutation ist in Zeit $\mathrm{O}\left(ln^{1/r}\right)$ möglich. Die Kommunikation am Ende von Phase i findet zwischen PEs statt, deren Position sich nur in einer Gitterkoordinate unterscheidet und die den Abstand $2^{\lfloor i/r \rfloor}$ zueinander haben. Der Zeitaufwand dafür ist in $\mathrm{O}\left(l2^{i/r}\right)$. Insgesamt ergibt sich ein Kommunikationsaufwand für alle Phasen in $\mathrm{O}\left(l\sum_{i<\log n} 2^{i/r}\right) = \mathrm{O}\left(ln^{1/r}\right)$. □

Der Rest des Beweises von Satz 7.14 läßt sich weitgehend analog zum Beweis von Satz 7.1 durchführen, wenn ein geeignetes $\Delta t \in \Theta\left(T_{\text{split}} + l\frac{n^{1/r}}{\log n}\right)$ gewählt wird. ■

Im Gegensatz zum Hyperwürfelalgorithmus wird nicht nur für kleine l eine Verbesserung gegenüber zufälligem Anfragen erzielt. Im Gegenteil, gerade wenn l groß ist, kann auch auf relativ kleinen Maschinen schon eine deutliche Beschleunigung auftreten. Die durch den Netzwerkdurchmesser verursachte Nachrichtenlatenz ist nämlich auf heutigen Maschinen oft vernachlässigbar gegenüber dem Nachrichtenaufsetzaufwand. Daß die nutzbare Kommunikations*bandbreite* bei Gittern von der Kommunikationsentfernung abhängt, ist dagegen für große l bereits bei kleinen n spürbar.

Auffällig ist, daß Δt um einen Faktor $\mathrm{O}(\log n)$ kleiner gewählt werden muß als der Kommunikationsaufwand für die letzten Phasen. Diese tragen deshalb kaum zum Fortschritt bei der sequentiellen Problembearbeitung bei. Dies tut der Analyse zwar keinen Abbruch, dennoch sei erwähnt, daß die letzten $r \log\log n$ Phasen jedes Zyklus weggelassen werden können,

[3)] In Praxis würde man trotzdem auf das randomisierte Verfahren aus Abschnitt 4.2 zurückgreifen, da die deterministischen Algorithmen auf Sortieren beruhen und deshalb recht komplex sind. Eine deterministische obere Schranke läßt sich trotzdem sicherstellen, indem das randomisierte Verfahren abgebrochen wird, wenn es zu lange dauert und erst dann auf Sortieren umgeschaltet wird.

ohne das grundsätzliche Verhalten des Algorithmus zu verändern. Es läßt sich leicht zeigen, daß dann in jeder Phase die sequentielle Berechnung über die Kommunikation überwiegt.

Schließlich sei kurz skizziert wie bei einem allgemeinen $d_1 \times \cdots \times d_r$-Gitter vorgegangen werden kann: Es wird ein $\lceil \log d_1 \rceil + \cdots + \lceil \log d_r \rceil$-dimensionaler Hyperwürfel so eingebettet, daß jedes PE mindestens einen, höchstens aber 2^r eingebettete Hyperwürfelknoten emuliert. Dies geschieht so, daß Dilation (Weglänge eingebetteter Kanten) und Kantenlast (Anzahl eingebetteter Kanten pro Gitterkante) höchstens um einen konstanten Faktor (1 bzw. 2) größer sind als in einem $2^{\lceil \log d_1 \rceil} \times \cdots \times 2^{\lceil \log d_r \rceil}$-Gitter. Dadurch wird die Kommunikation um einen konstanten Faktor langsamer und die sequentielle Rechengeschwindigkeit der virtuellen PEs schwankt um einen konstanten Faktor. Es läßt sich aber zeigen, daß empfängerveranlaßte Lastverteilungsalgorithmen solche Inhomogenitäten in den Rechen- und Kommunikationsgeschwindigkeiten „wegstecken“ können. Satz 7.14 gilt für beliebige Gitter wenn $d = d_1 + \cdots + d_r - r$ gesetzt wird und zwar weiterhin für beliebig kleine ε.

7.2.3 Fat trees

Bei Fat trees ist schon die naheliegendste Einbettung lokalitätserhaltend. Wird ein i-Würfel in einen Unterbaum mit 2^i PEs eingebettet, so hat dieser einen Durchmesser in $O(i)$. Leider ist $\sum_{i<\log n} i = (\log n)^2/2 + \Theta(1)$ und die Ausnutzung der Lokalität bringt nur einen Faktor zwei gegenüber einem Algorithmus bei dem in jeder Phase globale Kommunikation verwendet wird. Anders sieht es aus, wenn ein Zyklus schon nach $\sqrt{\log n}$ Phasen abgebrochen wird.

Satz 7.16. *Sei T_{perm} der Zeitbedarf für die Berechnung einer Zufallspermutation über $\mathbb{N}_n$. Wird Algorithmus 7.1 auf einem Fat tree eingesetzt und so modifiziert, daß ein Zyklus nur aus $\sqrt{\log n}$ Phasen besteht, so gibt es für alle Konstanten $\varepsilon > 0$ eine Wahl von $\Delta t \in \Theta\left(T_{\text{split}} + l + \sqrt{\log n} + \frac{T_{\text{perm}}}{\sqrt{\log n}}\right)$, so daß*

$$T_{\text{par}} \preceq (1 + \varepsilon\tilde{O}(1))\frac{T_{\text{seq}}}{n} + \tilde{O}\left(T_{\text{atomic}} + h(T_{\text{split}} + l + \sqrt{\log n} + \frac{T_{\text{perm}}}{\sqrt{\log n}})\right) \quad .$$

Beweis. Wieder ist der Beweis weitgehend analog zum Beweis von Satz 7.1. Im Gegensatz zum Hyperwürfel läßt sich die hohe Bandbreite des Fat trees hier nutzbringend einsetzen. Eine lokale Kommunikation nach einer Phase läßt sich in Zeit $\tilde{O}\left(\sqrt{\log n} + l\right)$ durchführen und für die globale Zufallspermutation ist der Zeitbedarf in $\tilde{O}(\log n + l)$. Diese Werte müssen entsprechend in die Berechnungen miteinbezogen werden. Die Beweis wird dadurch etwas einfacher, daß für hinreichend große n die $\log \frac{4}{\gamma}$ letzten Phasen des Hyperwürfelalgorithmus nicht betrachtet werden müssen, über die die Analyse wenig aussagen konnte. □

Fragen-und-Mischen auf Fat trees ist nur für kleine l (in $O\left(\sqrt{\log n}\right)$) besser als zufälliges Anfragen. Eine stärkere Aussage über den von T_{seq} abhängigen Term der parallelen Ausführungszeit wäre vermutlich möglich, würde aber ein tieferes Einsteigen in die Details des gewählten Datentransportverfahrens voraussetzen.

Etwas unangenehm ist der T_{perm}-Term in der parallelen Ausführungszeit, der bei Einsatz des Algorithmus aus Abschnitt 4.3.1 vor allem den Aufwand für eine Präfixsummenbildung repräsentiert. Das ist kein Problem, wenn die Maschine ein zusätzliches Baumnetzwerk hat, das Präfixsummen in Zeit $O(\log n)$ berechnen kann. Andernfalls ergäbe sich ein Aufwand in $O\left((\log n)^2\right)$. Vermutlich lassen sich die CRCW-PRAM-Algorithmen wie HAGERUP (1991) so für Fat trees modifizieren, daß $T_{\text{perm}} \in \log n \log^* n$.[4)] Ohne dies weiter auszuführen, sei hier aber ein noch besserer Ansatz skizziert: Permutationen werden nur ausgelöst, wenn mindestens $m \in \Omega(n)$ PEs arbeitslos sind (siehe auch Abschnitt 7.4). Um dies festzustellen, wird das Verfahren aus Abschnitt 4.2 eingesetzt, das ohne kollektive Operationen

[4)] $\log^* n := \min\left\{i \mid log^{(i)} n \leq 2\right\}$ mit $\log^{(0)} m := m$, $\log^{(k+1)} m := \log\log^{(k)} m$

auskommt. Der Algorithmus für Zufallspermutationen aus Abschnitt 4.3.1 wird so modifiziert, daß nur noch beschäftigte PEs ihre Nummer an einen zufälliges anderes PE schicken; dann werden die empfangenen Werte nur noch innerhalb eines Teilnetzwerks mit $2^{\sqrt{\log n}}$ PEs durchnumeriert und umverteilt. Das geht in Zeit $\tilde{O}(\log n)$ und für hinreichend große n erhält jedes PE mit hoher Wahrscheinlichkeit nur eine einzige PE-Nummer. Dadurch werden zwar keine gleichverteilten Zufallspermutationen erzeugt, aber vermutlich läßt sich relativ leicht zeigen, daß dies die Effizienz des Lastverteilungsverfahrens nicht beeinträchtigt.

7.3 Asynchrone Arbeitsweise

Algorithmus 7.1 ist als synchroner Algorithmus formuliert und diese Eigenschaft spielt in der Analyse auch eine wichtige Rolle. Eine globale Synchronisation nach jeder Phase ist aber auf Rechnern ohne Synchronisationshardware oder globale Uhr nicht tragbar. Der Synchronisationsaufwand könnte die Rechenzeit dominieren. Ein Ausweg beruht auf der Beobachtung, daß es genügt, PEs mit ihren jeweiligen Kommunikationspartnern zu synchronisieren. (Siehe auch (VOLLMAR, 1979, Abschnitt 3.1).) Es genügt also, nach Zeile 5 von Algorithmus 7.1 den Befehl „synchronize with PE $i_{\mathrm{PE}} \oplus 2^i$“ einzufügen, um auch eine asynchrone Ausführung der Schleifen zu ermöglichen. Eine asynchrone Arbeitsweise hat auch Vorteile – PEs können je nach ihrer individuellen Situation unterschiedliche Dinge tun. Von einem guten asynchronen Algorithmus würden wir erwarten, daß er diese Möglichkeit nutzt. Eine naheliegende Ergänzung ist z.B., daß PEs, die auf Synchronisation warten, weiter an ihrem Teilproblem arbeiten bis der Nachbar bereit ist. Auch auf die Auslieferung eines abgespaltenen Teilproblems muß nicht gewartet werden. Vor allem auf Gittern kann diese Überlappung von Kommunikation und Berechnung sinnvoll sein.

Interessanter ist die Frage, was ein PE macht, das arbeitslos ist und das noch eine vergleichsweise lange Wartezeit vor sich sieht, bis am Ende der laufenden Phase i wieder ein Lastausgleich stattfinden kann. Da in Satz 7.1 für kleine ε ein großes Δt gewählt werden muß, kann diese Wartezeit eine wichtige Quelle von Ineffizienz sein. Eine Möglichkeit ist die, eine Lastanfrage an einen Nachbarn entlang einer beliebigen (z.B. zufälligen) Dimension $j < i$ zu schicken. Da dadurch keine Abhängigkeiten zwischen verschiedenen i-Würfeln entstehen, wird die Analyse des Algorithmus dadurch nicht beeinflußt.

7.4 Einsparen von Permutationen

Die periodisch vorgenommenen Zufallspermutationen in Algorithmus 7.1 sind eine recht aufwendige Operation. Bei derjenigen Variante für Gitter, die die letzten Phasen eines Zyklus wegläßt, dominieren sie sogar den Kommunikationsaufwand. Das ist vor allem deshalb ärgerlich, weil es Probleminstanzen gibt, bei denen diese Mischoperationen nicht oder nur selten nötig sind. Besser wäre es, eine Durchmischung nur dann auszulösen, wenn sie tatsächlich nötig ist. Als Ausgangspunkt läßt sich der synchrone Algorithmus für zufälliges Anfragen aus Abbildung 6.1 verwenden. Dort wird eine Lastverteilung nur ausgelöst, wenn eine bestimmte Mindestzahl von PEs arbeitslos ist.

Die Anzahl arbeitsloser PEs punktuell am Ende eines Zyklus zu bestimmen, ist hier allerdings nicht aussagekräftig genug. Mehr Sinn macht es, die durchschnittliche PE-Auslastung während eines Zyklus zu bestimmen und eine Durchmischung auszulösen, wenn die Auslastung niedrig war. In der Praxis mag diese Maßnahme schon sehr erfolgreich sein. Die Analyse muß aber mit dem schlimmsten Fall rechnen. Und der könnte sein, daß jeweils ein Zyklus eine sehr gute PE-Auslastung aufweist, und dann ein Zyklus eine sehr schlechte. Da vor diesem schlecht ausgelasteten Zyklus keine Zufallspermutation vorgenommen wurde, kann nicht (wie in Lemma 7.5) garantiert werden, daß in schlecht ausgelasteten

Zyklen alle Teilprobleme gleichmäßig aufgespalten werden. Deshalb könnte eine Effizienz kleiner $\frac{1}{2}$ nicht ausgeschlossen werden.

var P : Subproblem
$P :=$ **if** $i_{\mathrm{PE}} = 0$ **then** P_{root} **else** $P_{\emptyset}$
$i :=$ idle $:= 0$
while not finished **do** synchronously
    $P :=$ work$(P, \Delta t)$
    if $T(P) = 0$ **then**
        idle := idle $+ 1$
        $(P, P\langle i_{\mathrm{PE}} \oplus 2^{i \bmod \log n}\rangle) :=$ split$(P\langle i_{\mathrm{PE}} \oplus 2^{i \bmod \log n}\rangle)$
    fi
    $i := i + 1$
    if $i \geq \log n \wedge i \bmod k \equiv 0$ **then**
        if $\sum_{j<n}$ idle$\langle j\rangle / i \geq m$ **then**
            permute subproblems randomly
            $i :=$ idle $:= 0$

Abbildung 7.4: Adaptives Fragen-und-Mischen.

Dieses Problem läßt sich lösen, indem die durchschnittliche PE-Auslastung mehrmals während eines Zyklus gemessen wird. Abbildung 7.4 zeigt eine Realisierung dieser Idee. Die grundsätzliche Vorgehensweise ist die gleiche wie in Algorithmus 7.1. In einer Variablen „idle“ zählt jedes PE nach wievielen Phasen seit der letzten Zufallspermutation es arbeitslos war. Sind seit der letzten Permutation mindestens $\log n$ Phasen verstrichen, so wird alle k Phasen die durchschnittliche Anzahl arbeitsloser PEs bestimmt. Nur wenn diese eine Schwelle m überschreitet, wird eine Zufallspermutation ausgelöst. Dieser adaptive Algorithmus benötigt für „gutmütige“ Probleminstanzen weniger Zufallspermutationen als der Grundalgorithmus. Außerdem läßt sich zeigen, daß er für schwierige Instanzen nicht schlechter geeignet ist:

Satz 7.17. *Es gibt für alle Konstanten $\varepsilon > 0$ eine Wahl von Δt, k und m, so daß für die Ausführungszeit von Algorithmus 7.4 gilt*

$$T_{\mathrm{par}} \preceq \left(1 + \varepsilon + \varepsilon\tilde{\mathrm{O}}\left(\frac{1}{\log\log n}\right)\right)\frac{T_{\mathrm{seq}}}{n} + \tilde{\mathrm{O}}\left(T_{\mathrm{atomic}} + h(l + T_{\mathrm{split}})\right) \quad .$$

Beweis. Der größte Teil des Beweises von Satz 7.1 läßt sich übernehmen, indem $m = \gamma n$ gewählt wird. Ein Zyklus hat nun eine variable Länge $\geq \log n$. Alle bis auf $\log \frac{4}{\gamma} + k$ Phasen eines Zyklus tragen entweder zur Zerkleinerung von Teilproblemen gemäß Lemma 7.6 bei, oder leisten (im Mittel) produktive Arbeit im Sinne von Lemma 7.7. Für ein geeignetes $k \in \Theta(\log n)$ läßt sich der Aufwand für die Berechnung von $\sum_{j<n}$ idle$\langle j\rangle$ auf die k Phasen verteilen, ohne daß T_{phase} sich dadurch asymptotisch erhöht. Durch eine einfache Rechnung läßt sich dann eine ähnliche Schranke für die parallele Ausführungszeit angeben wie in Relation (7.1).

$$T_{\mathrm{par}} \preceq \frac{\log n}{\log n - k - \log\frac{4}{\gamma}}\left(\frac{T_{\mathrm{seq}}}{n(1-\gamma)\Delta t} + \tilde{\mathrm{O}}(h) + \frac{T_{\mathrm{atomic}}}{\Delta t}\right)\left(\Delta t + T_{\mathrm{split}} + cl + \tilde{\mathrm{O}}\left(\frac{l}{\log\log n}\right)\right)$$

für eine geeignete Konstante c, $k \in \Theta(\log n)$ und hinreichend große n. Wird k geeignet gewählt, damit für hinreichend große n gilt $\frac{\log n}{\log n - k - \log \frac{4}{\gamma}} \leq \sqrt{1+\frac{\varepsilon}{2}}$ so kann der Rest des Beweises wiederum analog zum Beweis von Satz 7.1 erfolgen. Wird ε als Variable betrachtet, so ist $k \in \Theta\left((1+\frac{1}{\varepsilon})\log n\right)$ und $\Delta t \in \Theta\left((1+\frac{1}{\varepsilon})(T_{\text{split}} + l + \frac{1}{\varepsilon})\right)$.

□

Ein Schönheitsfehler dieses Ergebnisses ist, daß Δt quadratisch mit $\frac{1}{\varepsilon}$ wachsen muß. Denn für hohe Effizienz muß eine schlechte Auslastung nach $O(\varepsilon \log n)$ Phasen erkannt werden, und der Aufwand für diese Erkennung darf nur einen Anteil in $O(\varepsilon)$ einer Phase ausmachen. Dieses Problem tritt aber bei den asynchronen Algorithmusvarianten nicht mehr auf. Eine globale Summation benötigt nur $O(n)$ Operationen. Die meisten der $\Theta(n \log n)$ in der Analyse für die Summation veranschlagten Operationen können also für sequenzielle Berechnung genutzt werden.

Interessant ist in diesem Zusammenhang auch die in Abschnitt 4.2 beschriebene Technik der angenäherten globalen Summierung. Diese läuft völlig asynchron ab und benötigt ausschließlich Punkt-zu-Punkt Kommunikation, die sich in heutigen Systemen (z.B. MPI) leichter durch sequentielle Berechnung überlappen läßt als kollektive Operationen.

7.5 Kombination mit zufälligem Anfragen

Die in oben beschriebenen Verfeinerungen von Fragen-und-Mischen für asynchrone Arbeitsweise und zum Einsparen von Permutationen, lassen sich mit zufälligem Anfragen zu einer interessanten Familie von neuen Algorithmen kombinieren. *Lokales zufälliges Anfragen* sei hier am Beispiel von Gittern beschrieben, läßt sich aber auch an andere Netzwerke anpassen, bei denen lokale Kommunikation in Teilnetzen deutlich effizienter als globale Kommunikation ist.

Als Ausgangspunkt sei der am Ende von Abschnitt 7.2.2 beschriebene Algorithmus betrachtet, bei dem die letzten $r \log\log n$ Phasen weggelassen werden. Dazu wird die am Ende von Abschnitt 7.3 beschriebene Idee aufgegriffen, daß PEs, die arbeitslos werden, noch vor Phasenende Anfragen innerhalb ihres i-Würfels schicken können. Bei Gittern gibt es keinen Grund, diese Anfragen auf Nachbarn im eingebetteten Hyperwürfel zu beschränken. Statt dessen kann zufälliges Anfragen lokal, innerhalb des i-Würfels verwendet werden. Innerhalb dieses i-Würfels gilt dann die Analyse von zufälligem Anfragen, die garantiert, daß lokal eine gleichmäßige Lastverteilung stattfindet.

Diese Maßnahme ergänzt sich mit den in Abschnitt 7.4 beschriebenen Methoden zur Einsparung von Permutationen. Ist Phase $\log n - r \log\log n$ erreicht, so kann weiterhin innerhalb des Teilnetzes der Größe $\frac{n}{(\log n)^r}$ lokales zufälliges Anfragen durchgeführt werden, solange die PE-Auslastung gut ist. Dann sind auch keine Synchronisationen mehr nötig. Bei sorgfältiger Formulierung der Modifikationen bleibt der Lastverteiler mindestens so effizient wie der Grundalgorithmus für Fragen-und-Mischen. Bei einfacheren Probleminstanzen kann aber viel Aufwand eingespart werden. Zum Beispiel wird sich in Abschnitt 8.4.2 zeigen, daß im α-Spaltmodell auf Zufallspermutationen u.U. ganz verzichtet werden kann.

Es gibt eine Vielzahl von weiteren möglicherweise nützlichen Verbesserungen, von denen allerdings nicht unbedingt klar ist, ob das Verhalten im schlimmsten Fall dadurch beeinträchtigt wird. Hier seien nur einige aufgezählt:

- Lokales zufälliges Anfragen nicht nur innerhalb disjunkter Teilgitter durchführen, sondern in einem festen Umkreis jedes PE. Dadurch lassen sich Lastunterschiede zwischen Teilnetzen u.U. auch ohne Zufallspermutationen ausgleichen.
- Mit Wahrscheinlichkeit $\Theta\left(\frac{1}{\log n}\right)$ wird eine globaler Lastanfrage eingestreut. Selbst ohne Zufallspermutationen ist das Verfahren dann asymptotisch wenigstens so schnell wie zufälliges An-

fragen; selbst wenn alle lokalen Anfragen vergeblich sind, fallen die Kosten dafür asymptotisch nämlich nicht ins Gewicht. Umgekehrt dominieren auch die globalen Anfragen nicht die Kommunikationskosten.

- Synchronisation und Phasen ganz weglassen. Dann bleibt zufälliges Anfragen mit lokaler Kommunikation, die durch gelegentliche Mischoperationen „klumpenfrei“ gehalten wird.
- Ersatz echter Permutationen durch einfacher zu berechnende Pseudozufallspermutationen. Insbesondere kann bei der obigen phasenlosen Variante ausgenutzt werden, daß es keine Rolle spielt, wo innerhalb eines Teilnetzes ein Teilproblem plaziert wird. Es kommt nur darauf an, in welchem Teilnetz es landet.
- Mischoperationen nicht nur global durchführen, sondern je nach Situation nur zwischen einigen Teilnetzen. Zum Beispiel könnte das Umschalten zwischen Phasen ebenfalls adaptiv durch Beobachtung der Auslastung der Teilnetze geschehen. Sind zwei i-Würfel, die einen $i+1$-Würfel bilden, unterschiedlich gut ausgelastet, werden die Teilprobleme der beiden i-Würfel durchmischt, die aktuelle Phase wird auf $i+1$ erhöht, und der benachbarte $i+1$-Würfel wird informiert, damit auch dieser auf Phase $i+1$ umschaltet. (Wegen der Komplexität dieses Protokolls mag es Sinn machen, gleich mehrere Phasen weiterzuschalten.)

7.6 Zusammenfassung

Durch synchronisierte Kommunikation entlang der Kanten eines (eingebetteten) Hyperwürfels kann eine ähnlich gleichmäßige Aufspaltung von Teilproblemen erreicht werden, wie bei zufälligem Anfragen, wenn die Teilprobleme oft genug „umgerührt“ werden. Ein hinter dieser Idee stehendes allgemeines Prinzip könnte darin bestehen, daß nach einer zufälligen Vertauschung auch Anfragen an deterministisch berechnete PEs zufällige Teilprobleme erreichen, wenn sichergestellt ist, daß die beteiligten PEs seit der Vertauschung nicht interagiert haben.

Aus dieser Erkenntnis ergibt sich die Möglichkeit, empfängerveranlaßte Lastverteilungsalgorithmen zu bauen, die mit lokaler Kommunikation (bei hyperwürfelartigen Netzen) oder doch mit weniger als globaler Kommunikation auskommen. Dies bewirkt nicht nur eine Verringerung der Nachrichtenlatenz, sondern auf vielen Netzwerken auch eine Vergrößerung der nutzbaren Bandbreite.

Hier wurden Gitter nur als ein wichtiges und repräsentatives Beispiel herausgegriffen. Die Idee der Mischung von globaler und lokaler Kommunikation ist auch für die in Abschnitt 2.2.1 beschriebenen hierarchischen hybriden Netzwerke wichtig. Nur sind asymptotische Aussagen schwierig, weil keine Einigkeit herrscht, wie solche Architekturen für beliebiges n zu skalieren sind. Das Verfahren läßt sich sogar weitgehend portabel implementieren, wenn eine Prozessornumerierung verwendet wird, bei der nah beieinander liegende PEs nah beienander liegende PE-Nummern haben. Für Gitter ist z.B. die Hilbert-Numerierung gut geeignet (NIEDERMEIER UND SANDERS, 1996).

Wie die Diskussion in Abschnitt 7.5 zeigt, gibt es eine Vielzahl von Ansätzen, den Grundalgorithmus zu verbessern. Da einige vom praktischen Standpunkt verlockende Modifikationen die Analyse schwieriger machen, ergibt sich hier ein weites Feld für theoretische und experimentelle Untersuchungen, die aber den Rahmen dieser Arbeit sprengen würden.

Ergebnisse

Die hier beschriebenen Algorithmen sind die asymptotisch effizientesten bekannten Lastverteilungsalgorithmen für baumförmige Berechnungen. Es sind die ersten effizienten empfän-

gerveranlaßten Algorithmen, die nicht hauptsächlich global kommunizieren. Interessant ist ein Vergleich mit den in LEIGHTON ET AL. (1989); RANADE (1994) beschriebenen randomisierten dynamischen Baumeinbettungsalgorithmen für hyperwürfelartige Netzwerke. Diese sind in der Lage, einen a priori unbekannten Binärbaum mit m Knoten mit hoher Wahrscheinlichkeit mit asymptotisch optimaler maximaler Dilation, Kantenlast und Knotenlast einzubetten. Das ergibt aber nur einen effizienten Baumsuchalgorithmus, wenn die pro Suchbaumknoten anfallende Berechnung groß gegen die Zeit für eine Teilproblemübertragung ist. Selbst dann ist keine Effizienz nahe 1 erreichbar. Umgekehrt läßt sich aus der Idee von Fragen-und-Mischen ein dynamischer Baumeinbettungsalgorithmus entwickeln, der im *Mittel* eine konstante Dilation erreicht.

Kapitel 8

Statische Lastverteilung

Schon in Abschnitt 2.4.1 wurde darauf hingewiesen, daß die Lastverteilung sich mit minimaler Kommunikation – durch eine einzige Broadcast Operation – realisieren läßt. Das ist attraktiv, vor allem wenn Kommunikation teuer ist. Die in Abschnitt 5.3 hergeleiteten Schranken zeigen jedoch genauso wie die praktische Erfahrung, daß die dadurch erzielte Lastverteilung bei Baumsuchproblemen zu schlecht ist. Die Situation könnte jedoch günstiger sein, wenn mehr Teilprobleme erzeugt werden, als es PEs gibt. Wird die Zuordnung zufällig durchgeführt, so könnte eine sehr ungleichmäßige Verteilung u.U. unwahrscheinlich werden.

In diesem Kapitel wird genauer untersucht, in welchen Fällen diese Idee erfolgversprechend ist. Der größte Teil der Analyse wird für ein abstrakteres und allgemeineres Modell durchgeführt, das in Abschnitt 8.1 eingeführt wird. Früher verwendete Ansätze zur Modellierung statischer Lastverteilung haben nämlich Schwächen, die für viele Anwendungen gelten und bei baumförmigen Berechnungen nur besonders schwerwiegend sind. In Abschnitt 8.2 wird das allgemeine Modell analysiert. Erst in Abschnitt 8.3 wird dann diskutiert, wie sich diese Ergebnisse für baumförmige Berechnungen auswirken. Es ergeben sich auch Konsequenzen für Entwurf und Analyse dynamischer Lastverteilungsalgorithmen. In Abschnitt 8.4 wird z.B. gezeigt, daß statische Lastverteilung in Kombination mit einer Variante von zufälligem Anfragen in einigen Fällen eine asymptotische Verbesserung gegenüber den Einzelverfahren erzielen kann. Abschnitt 8.5 faßt die Ergebnisse zusammen und diskutiert zusätzliche Anwendungsmöglichkeiten von randomisierter Baumzerlegung.

8.1 Ein abstraktes Modell

Es wird in Abschnitt 8.1.1 zunächst ein kurzer Überblick über bisherige Ansätze zur Modellierung statischer Lastverteilung gegeben. Die dabei zutage tretenden Probleme sind Anlaß für das in Abschnitt 8.1.2 eingeführte spieltheoretische Modell, bei dem die Teilproblemgröße von einem Gegner oder Gegenspieler (adversary) festgelegt wird.

8.1.1 Frühere Ansätze

Ein typisches Beispiel für die Modellierung statischer Lastverteilung von Teilproblemen unbekannter Größe ist KRUSKAL UND WEISS (1985). Dort wird davon ausgegangen, daß Teilproblemgrößen identisch verteilte unabhängige Zufallsvariablen mit einer Eigenschaft namens „increasing failure rate“ sind. Diese besagt u.a., daß große Abweichungen vom Mittelwert in gewissem Sinne vernachlässigbar sind. Für dieses Modell wird dann gezeigt, daß es genügt, $O(\log n)$ Teilprobleme pro PE zu allozieren, um im Mittel eine gute Lastverteilung zu erhalten.

Für Anwendungen wie Monte-Carlo Simulation (HEIDELBERGER, 1988) kann dies ein gutes Problemmodell sein. Im allgemeinen sind Teilproblemgrößen jedoch keine Zufallsvariablen, sondern werden durch einen komplizierten u.U. deterministischen aber unbekannten Prozeß bestimmt. Der Zufall kommt erst dadurch ins Spiel, daß Teilprobleme zufällig auf die PEs verteilt werden. Dieses Problem wird oft dadurch umgangen, daß ein stochastischer Prozeß zur Erzeugung der Teilprobleme postuliert wird, der zwar nicht der realen Anwendung entspricht, aber bei geeigneter Modellierung interessante Aussagen zuläßt. Für baumförmige Berechnungen wird ein solcher Ansatz z.B. in SANDERS (1994f) verfolgt. Dort wird die Spaltoperation durch eine Zufallsvariable X mit $\mathbf{Bild}(X) = [0,1]$ und $\mathbf{E}X = \frac{1}{2}$ modelliert. Ein Teilproblem der Größe t wird in Teilprobleme der Größe Xt und $(1-X)t$ gespalten. Verschiedene Spaltoperationen sind unabhängig voneinander. Es werden dann Aussagen über die Qualität statischer Lastverteilung in Abhängigkeit von der Standardabweichung von X gemacht.

Selbst in diesem Modell sind Teilproblemgrößen aber voneinander abhängig, da alle Teilprobleme von einem gemeinsamen Vorfahren abgeleitet sind. Die Teilproblemgrößen sind außerdem sehr ungleichmäßig verteilt, so daß große Abweichungen vom Mittelwert entscheidend für die Analyse sind. Es zeigt sich, daß es im allgemeinen keineswegs ausreicht, logarithmisch viele Teilprobleme pro PE zu allozieren.

8.1.2 Ein spieltheoretisches Modell

Gegenstand des Spiels sind m Teilprobleme P_i ($i \in \mathbb{N}_m$) mit Größe $t_i = T(P_i)$. Teilprobleme können unabhängig voneinander auf verschiedenen PEs abgearbeitet werden. Die Teilproblemgrößen seien normiert, so daß $\sum_{i<m} t_i = 1$. Außerdem sei $t_{\max}$ eine obere Schranke für eine Teilproblemgröße. Um eine Anzahl uninteressanter Spezialfalldiskussionen zu vermeiden, sei angenommen, daß $n > 1$, $n|m$, $t_{\max} \leq 1/n$ und $1/t_{\max} \in \mathbb{N}$. Ein Spielpartner ist ein (potentiell randomisierter) Lastverteilungsalgorithmus. Dieser legt ohne Kenntnis der Teilproblemgrößen eine Zuordnung $A \in \mathbb{N}_n^{\mathbb{N}_m}$ fest, so daß P_i auf PE $A(i)$ abgearbeitet wird. Der Gegner (adversary) legt die Teilproblemgrößen unter Beachtung der oben gegebenen Bedingungen fest. Der Gegner kennt den Lastverteilungsalgorithmus, nicht aber die Zufallsbits, die der Lastverteiler verwendet. $L_i := \sum_{\{j|A(j)=i\}} t_j$ bezeichne die Last von PE i. Ziel des Lastverteilers ist es, die maximale Last $L_{\max} := \max_{i<n} L_i$ zu minimieren. Ziel des Gegners ist es, $L_{\max}$ zu maximieren. Zu gegebenem ε hat der Lastverteiler genau dann gewonnen, wenn $L_{\max} \leq \frac{1+\varepsilon}{n}$. Ein Lastverteiler, der mit der Last fertig wird, die ein optimal spielender Gegner erzeugt, ist offensichtlich für alle tatsächlichen Berechnungen geeignet, sofern diese nicht von Zeit und Ort der Teilproblembearbeitung abhängen.

8.2 Analyse

Dieser Abschnitt analysiert zwei randomisierte Strategien des Lastverteilers. In Abschnitt 8.2.1 wird mit einer mathematisch besonders einfachen Strategie begonnen. Für diese

wird zunächst eine Klasse von gegnerischen Strategien identifiziert, die in gewissem Sinne den schlechtesten Fall darstellen. Dann wird analysiert, unter welchen Umständen der Lastverteiler trotzdem mit hoher Wahrscheinlichkeit einen guten Lastausgleich erreichen kann. Daraus lassen sich (zumindest) für das Verhalten im Mittel Schlüsse über das Verhalten des Lastverteilers gegenüber beliebigen Gegnerstrategien ziehen. In Abschnitt 8.2.2 wird dann eine vom Standpunkt der Parallelverarbeitung sinnvollere Lastverteilungsstrategie eingeführt und gezeigt, daß diese mindestens so effektiv ist wie die vorher betrachtete.

8.2.1 Lastverteilungsstrategie unabhängiges Zuordnen

Es wird nun die folgende einfache randomisierte Lastverteilungsstrategie näher betrachtet:

Definition 8.1. In der Lastverteilungsstrategie *unabhängiges Zuordnen* werden die Teilprobleme unabhängig voneinander einem zufällig gleichverteilt gewählten PE zugeordnet.

Anders ausgedrückt, wird ein zufälliges Element aus der Menge

$$\Omega = \mathbb{N}_n^{\mathbb{N}_m}$$

aller möglichen Teilproblemzuordnungen ausgewählt.

Eine Schwierigkeit bei der Analyse besteht darin, daß es überabzählbar viele, z.T. sehr komplexe Strategien des Gegners gibt. Ziel ist es, zu zeigen, daß unabhängiges Zuordnen mit all diesen gegnerischen Strategien zurechtkommt. Es wird sich herausstellen, daß es genügt, das Verhalten des Lastverteilers gegenüber einer besonders einfach aufgebauten Klasse von gegnerischen Strategien zu analysieren. Es ist intuitiv einleuchtend, daß die Lastverteilung besonders schwierig ist, wenn die Teilproblemgrößen so verschieden wie möglich sind. Daraus ergibt sich die folgende Klasse von Strategien für den Gegner:

Definition 8.2. Eine Teilproblemgrößenzuordnung des Gegners heißt *Max-Strategie*, wenn alle Teilproblemgrößen entweder 0 oder $t_{\max}$ sind.

Der folgende Satz zeigt, daß Max-Strategien in gewissem Sinne optimale gegnerischen Strategien gegen unabhängiges Zuordnen sind. Max-Strategien maximieren das erwartete Lastungleichgewicht.

Satz 8.3. *Zu jeder Strategie des Gegners gibt es eine Max-Strategie des Gegners, die* $\mathbf{E}L_{\max}$ *nicht vermindert.*

Ein großer Teil des Beweises von Satz 8.3 läßt sich auf die entsprechende Aussage über deterministische Gegnerstrategien zurückführen:

Lemma 8.4. *Zu jeder deterministischen Strategie des Gegners gibt es eine deterministische Max-Strategie des Gegners, die* $\mathbf{E}L_{\max}$ *nicht vermindert.*

Beweis. Gegeben sei eine beliebige deterministische Strategie des Gegners. Falls diese Strategie kein Teilproblem i mit $0 < t_i < t_{\max}$ festlegt, folgt die Behauptung. Andernfalls gibt es (wegen $\frac{1}{t_{\max}} \in \mathbb{N}$) ein zweites Teilproblem j mit $0 < t_j < t_{\max}$. O.B.d.A. sei $t_i \geq t_j$.

Die Zufallsvariable $L_{\max}$ läßt sich als Abbildung $L_{\max} \in \mathbb{R}^{\Omega}$ vom Merkmalsraum Ω der Menge aller Zuordnungen in die reellen Zahlen auffassen. Nun wird Ω folgendermaßen partitioniert:

$$\Omega = \Omega_{ij} \dot\cup \Omega_{i\bar{j}} \dot\cup \Omega_{\bar{i}j} \dot\cup \Omega_{\bar{i}\bar{j}}$$

mit

$\Omega_{ij} :=$ Die Zuordnungen, die sowohl i als auch j Maximumstellen von L zuordnen. (Also PEs k_i und k_j mit $L_{k_i} = L_{k_j} = L_{\max}$).

$\Omega_{i\bar{j}} :=$ Die Zuordnungen, die i aber nicht j einer Maximumstelle von L zuordnen.

$\Omega_{\bar{i}j} :=$ Die Zuordnungen, die j aber nicht i einer Maximumstelle von L zuordnen.

$\Omega_{\bar{i}\bar{j}} :=$ Die Zuordnungen, die weder i noch j Maximumstellen von L zuordnen.

Nach Definition des Erwartungswertes gilt

$$\mathbf{E}L_{\max} = \frac{\sum\limits_{\omega\in\Omega_{ij}} L_{\max}(\omega) + \sum\limits_{\omega\in\Omega_{i\bar{j}}} L_{\max}(\omega) + \sum\limits_{\omega\in\Omega_{\bar{i}j}} L_{\max}(\omega) + \sum\limits_{\omega\in\Omega_{\bar{i}\bar{j}}} L_{\max}(\omega)}{|\Omega|} \quad . \tag{8.1}$$

Sei $\Delta := \min(t_j, t_{\max} - t_i)$. Der Gegner ersetze nun t_i durch $t_i' := t_i + \Delta$ und t_j durch $t_j' := t_j - \Delta$. Abbildung 8.1 verdeutlicht die möglichen Konsequenzen; t_i' wird zu t_{max} oder t_j' wird 0. $\mathbf{E}L_{\max}$ ändert sich folgendermaßen:

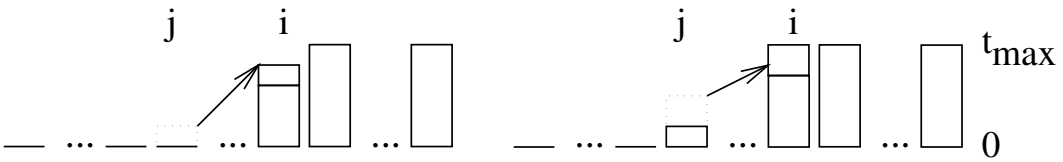

Abbildung 8.1: Mögliche Auswirkungen, wenn t_i und t_j um $\Delta = \min(t_j, t_{\max} - t_i)$ verändert werden.

- $\sum_{f\in\Omega_{ij}} L_{\max}(f)$ kann nur größer werden: Summanden bleiben gleich, wenn i und j der gleichen Maximumstelle zugeordnet werden. Wenn i und j verschiedenen Maximumstellen zugeordnet werden, erhöht sich der Summand um Δ.
- $\sum_{f\in\Omega_{i\bar{j}}} L_{\max}(f)$ erhöht sich um $|\Omega_{i\bar{j}}|\Delta$ weil jeder einzelne Summand sich um Δ erhöht.
- $\sum_{f\in\Omega_{\bar{i}j}} L_{\max}(f)$ verringert sich um höchstens $|\Omega_{\bar{i}j}|\Delta$.
- $\sum_{f\in\Omega_{\bar{i}\bar{j}}} L_{\max}(f)$ kann nur größer werden.

Wegen $t_i \geq t_j$ gilt aber $|\Omega_{i\bar{j}}| \geq |\Omega_{\bar{i}j}|$. Dies läßt sich z.B. leicht einsehen indem beachtet wird, daß die Abbildung „Vertausche die Positionen der Teilprobleme i und j“ eine injektive Abbildung von $\Omega_{\bar{i}j}$ nach $\Omega_{i\bar{j}}$ ist.[1)] Insgesamt kann $\mathbf{E}L_{\max}$ sich durch die Abänderung der Größenzuordnung also nur vergrößern.

[1)] Ich möchte Thomas Worsch für den Hinweis danken, daß auch hier injektive Funktionen helfen können, die Mächtigkeit von Mengen zu vergleichen (wie schon in im Beweis von Lemma 7.4).

Die Abänderung von Teilproblemgrößen kann solange wiederholt werden, bis es keine nichtextremen Teilproblemgrößen mehr gibt. Dazu genügen endlich viele Schritte, da mit jedem Schritt eine Teilproblemgröße auf 0 oder $t_{\max}$ gesetzt wird. Folglich gibt es eine Max-Strategie, die $\mathbf{E}L_{\max}$ nicht vermindert. □

Beweis von Satz 8.3. Eine randomisierte Strategie des Gegners läßt sich als zufällige Auswahl zwischen deterministischen Strategien auslegen. Jede dieser Teilstrategien kann – unabhängig von den anderen – mit der im Beweis von Lemma 8.4 beschriebenen Technik zu einer Max-Strategie gemacht werden, ohne $\mathbf{E}L_{\max}$ zu vermindern. Eine zufällige Auswahl zwischen Max-Strategien ist aber selbst eine Max-Strategie, die $\mathbf{E}L_{\max}$ nicht vermindert. □

Nachdem Max-Strategien sich als schwierig zu balancieren herausgestellt haben, soll nun untersucht werden, unter welchen Umständen der Lastverteiler trotzdem erfolgreich sein kann:

Satz 8.5. *Für jedes* $\varepsilon > 0$*, jedes* $\beta > 0$ *und jede Max-Strategie des Gegners gibt es eine Konstante c, so daß* $\mathbf{P}\left[L_{\max} > \frac{1+\varepsilon}{n}\right] \leq n^{-\beta}$ *falls* $t_{\max} \leq \frac{1}{cn\ln n}$.

Beweis. Da alle PEs gleich behandelt werden, genügt es, ein geeignetes c zu bestimmen, so daß $\mathbf{P}\left[L_j > \frac{1+\varepsilon}{n}\right] \leq n^{-(\beta+1)}$ für ein beliebiges j (siehe auch Gleichung (3.3)). Da Teilprobleme der Größe Null nicht zu $L_{\max}$ beitragen, genügt es außerdem, nichtleere Teilprobleme zu betrachten. Im Fall einer Max-Strategie gibt es genau $\frac{1}{t_{\max}}$ nichtleere Teilprobleme, und diese haben alle die Größe $t_{\max}$. Sei nun

$$Y_{ji} := [\text{Das } i\text{-te nichtleere Teilproblem wird PE } j \text{ zugeordnet.}] \ .$$

Dann ist

$$L_j = t_{\max} \sum_{i < \frac{1}{t_{\max}}} Y_{ji} \ .$$

Da die Teilprobleme unabhängig voneinander und zufällig gleichverteilt den PEs zugeordnet werden, sind die Y_{ji} für festes j unabhängige 0/1-Zufallsvariablen mit $\mathbf{P}[Y_{ji} = 1] = \frac{1}{n}$. Mit Hilfe der Chernoff-Schranke 3.5 für kleine positive Abweichungen vom Erwartungswert kann deshalb die folgende Abschätzung gemacht werden:

$$\mathbf{P}\left[L_j > \frac{1+\varepsilon}{n}\right] \leq e^{-\frac{\varepsilon^2}{2nt_{\max}}} \leq e^{-\frac{\varepsilon^2 cn\ln n}{2n}} = n^{-\frac{\varepsilon^2 c}{2}} \leq n^{-(\beta+1)}$$

für $c \geq 2(\beta+1)/\varepsilon^2$. □

Wenn die maximale Teilproblemgröße also hinreichend klein ist, damit es mindestens $cn\log n$ Teilprobleme geben muß, ist eine erfolgreiche Lastverteilung hochwahrscheinlich. Leider folgt daraus aber noch nicht, daß dies für beliebige gegnerische Strategien mit hoher Wahrscheinlichkeit gilt. Für das Verhalten im Mittel ist der Schluß auf beliebige Gegnerstrategien jedoch mit Hilfe von Satz 8.3 möglich:

Satz 8.6. *Für jedes* $\varepsilon > 0$ *und jede Strategie des Gegners gibt es eine Konstante c, so daß* $\mathbf{E}L_{\max} \leq \frac{1+\varepsilon}{n}$ *falls* $t_{\max} \leq \frac{1}{cn\ln n}$.

Beweis. Nach Satz 8.3 genügt es, Max-Strategien zu betrachten, da andere Strategien $\mathbf{E}L_{\max}$ nicht vergrößern können. Nach Satz 8.5 gibt es eine Wahl von c so daß $\mathbf{P}\left[L_{\max} > \frac{1+\varepsilon/2}{n}\right] \leq \frac{\varepsilon/2}{n}$. (Hier wird die Annahme $n > 1$ aus Abschnitt 8.1.2 benötigt.) Außerdem gilt $L_{\max} \leq \sum_{i<m} t_i = 1$. Aus diesen Tatsachen ergibt sich die folgende Abschätzung:

$$\begin{aligned}\mathbf{E}L_{\max} &\leq \frac{1+\varepsilon/2}{n}\mathbf{P}\left[L_{\max} \leq \frac{1+\varepsilon/2}{n}\right] + 1 \cdot \mathbf{P}\left[L_{\max} > \frac{1+\varepsilon/2}{n}\right] \\ &\leq \frac{1+\varepsilon/2}{n} \cdot 1 + 1 \cdot \frac{\varepsilon/2}{n} = \frac{1+\varepsilon}{n}\end{aligned}$$

□

8.2.2 Lastverteilungsstrategie zufälliges Verteilen

Die Strategie des unabhängigen Zuordnens erscheint vom algorithmischen Standpunkt ungünstig, da i. allg. nicht alle PEs gleichviele Teilprobleme abbekommen. Deshalb liegt es nahe, die möglichen Zuordnungen so einzuschränken, daß dies garantiert ist:

Definition 8.7. In der Lastverteilungsstrategie *zufälliges Verteilen* wird gleichverteilt ein zufälliges Element der Menge

$$\bar{\Omega} := \left\{ \omega \in {\mathbb{N}_n}^{\mathbb{N}_m} \mid \forall k < n : \left|\omega^{-1}(k)\right| = \frac{m}{n} \right\}$$

ausgewählt.

Diese Variante läßt sich auch als Urnenproblem ohne Zurücklegen auffassen:

Lemma 8.8. *Definition 8.7 ist gleichbedeutend damit, daß jedes PE $\frac{m}{n}$ Teilprobleme aus einer Urne ohne Zurücklegen zieht (in beliebiger Reihenfolge).*

Beweis. Sei $\omega \in \bar{\Omega}$ beliebig. Jedes PE hat $\frac{m}{n}!$ Möglichkeiten „seine“ durch ω festgelegten Teilprobleme auszuwählen – unabhängig von der Reihenfolge in der die PEs aus der Urne ziehen. Die Anzahl der Möglichkeiten hängt also nicht von ω ab und folglich wird jedes $\omega \in \bar{\Omega}$ mit gleicher Wahrscheinlichkeit gewählt. Da auch keine Zuordnungen gewählt werden können, die nicht in $\bar{\Omega}$ liegen, folgt die Behauptung. □

Im folgenden wird gezeigt, daß die Aussagen aus Abschnitt 8.2.1 über gleichmäßiges Zuordnen auch für zufälliges Verteilen gelten. Was die Optimalität von Max-Strategien betrifft, so gilt die Aussage von Satz 8.3 sogar für jede Verteilung, die invariant bezüglich der Vertauschung von Teilproblempositionen ist:

Satz 8.9. *Satz 8.3 gilt auch für alle Verteilungen auf der Menge Ω, bei denen gilt $\mathbf{P}[\omega] = \mathbf{P}[\omega']$ wann immer ω und ω' Zuordnungen sind, die sich nur durch Vertauschen zweier Teilproblempositionen unterscheiden.*

Beweis. Weitgehend analog zum Beweis von Satz 8.3. Wird eine beliebige Verteilung auf der Menge aller Zuordnungen betrachtet, so muß in Gleichung (8.1) $L_{\max}(\omega)$ jeweils durch $\mathbf{P}[\omega]\,L_{\max}(\omega)$ ersetzt werden. Es bleibt dann zu zeigen, daß $\sum_{f\in\Omega_{i\bar{j}}} \mathbf{P}[\omega] \geq \sum_{f\in\Omega_{\bar{i}j}} \mathbf{P}[\omega]$. Die

Injektion von $\Omega_{\bar{i}j}$ nach $\Omega_{i\bar{j}}$, die die Positionen von i und j vertauscht, definiert zu jedem Element ω von $\Omega_{\bar{i}j}$ ein eindeutiges Element ω' von $\Omega_{i\bar{j}}$, das nach Voraussetzung mit gleicher Wahrscheinlichkeit gewählt wird. Folglich enthält $\sum_{f\in\Omega_{i\bar{j}}} \mathbf{P}[\omega]$ mindestens die gleichen Glieder wie $\sum_{f\in\Omega_{\bar{i}j}} \mathbf{P}[\omega]$. □

Satz 8.10. *Die Sätze 8.3, 8.5 und 8.6 gelten auch für zufälliges Verteilen.*

Beweis. Das Gegenstück zu Satz 8.3 ist ein einfaches Korollar zu Satz 8.9. Der Beweis von Satz 8.5 beruht auf einer Abschätzung der Wahrscheinlichkeit, daß die Anzahl nichtleerer Teilprobleme für PE j das $(1+\varepsilon)$-fache ihres Erwartungswertes übersteigt. Die Anzahl nichtleerer Teilprobleme läßt sich auffassen als die Anzahl weißer Kugeln, die bei $\frac{m}{n}$-maligem Ziehen ohne Zurücklegen aus einer Urne mit m gezogen werden, von denen $\frac{1}{t_{\max}}$ weiß sind. Nach Lemma 8.8 ergibt sich bei zufälligem Verteilen die gleiche Situation, nur daß ohne Zurücklegen gezogen wird. Nach Lemma 3.3 ist deshalb die im Beweis von Satz 8.5 ausgerechnete Wahrscheinlichkeit eine konservative Abschätzung für die hier zu betrachtende Wahrscheinlichkeit. Der Beweis von Satz 8.6 gilt damit auch für zufälliges Verteilen. □

8.3 Anwendung auf Baumsuche

Das abstrakte Konzept der zufälligen Verteilung von m Teilproblemen aus Abschnitt 8.1 läßt sich auf baumförmige Berechnungen anwenden, indem das Wurzelproblem in m Teile zerteilt wird. In Abschnitt 8.3.1 wird gezeigt, wie sich dieser Ansatz effizient umsetzen läßt. Dieser Algorithmus wird dann in Abschnitt 8.3.2 analysiert.

8.3.1 Der Algorithmus

Es können $m = 2^{h'}$ Teilprobleme erzeugt werden, indem das Wurzelproblem rekursiv h' mal hintereinander aufgespalten wird. Es sei davon ausgegangen, daß $n \leq 2^{h'}$ eine Zweierpotenz ist. Dann gilt es, jedem PE $\frac{2^{h'}}{n}$ Teilprobleme zufällig zuzuordnen.[2)] Es wird sich herausstellen, daß sehr viele Teilprobleme erzeugt und verteilt werden müssen. Deshalb ist es entscheidend, diese „unproduktive“ Arbeit schnell zu erledigen. Die naheliegende sequentielle Erzeugung von Teilproblemen ist deshalb nicht sinnvoll. Selbst eine parallele Erzeugung mit anschließendem Austausch der Teilprobleme wäre sehr aufwendig.

Statt dessen wird der in Abschnitt 2.4.1 eingeführte Ansatz des mehrfachen Spaltens des Wurzelproblems erweitert. Jedes PE erzeugt genau die Teilprobleme, die es auch bearbeitet und zwar eins nach dem anderen, so daß Speicherplatz für konstant viele Teilprobleme pro PE ausreicht. Dazu genügt es, daß jedes PE das Wurzelproblem, seine Nummer und eine globale Permutation $\pi \in \mathbb{N}_{2^{h'}}{}^{\mathbb{N}_{2^{h'}}}$ kennt. Die zu erzeugenden Teilprobleme lassen sich als die Blätter eines vollständigen geordneten binären Spaltbaumes der Tiefe h' auffassen. Das Wurzelproblem P_{root} ist die Wurzel. Ist P ein innerer Knoten des Baums, so sind P_0 und P_1

[2)] Der Algorithmus läßt sich leicht so verallgemeinern, daß beliebige n möglich sind: Teile jedem PE zufällig $\left\lfloor \frac{2^{h'}}{n} \right\rfloor$ Teilprobleme zu. Die verbleibenden $\leq n$ Teilprobleme können so verteilt werden, daß kein PE mehr als eines davon erhält. Das dadurch verursachte zusätzliche Ungleichgewicht ist klein, da ohnehin nur für $\frac{2^{h'}}{n} \gg 1$ eine vernünftige Lastverteilung zu erwarten ist.

```
var     P, P'    : Subproblem
        h'       : Integer                          (* splitting depth *)
        π        : ℕ_{2^h'}^{ℕ_{2^h'}}
Broadcast root problem P_root
Determine a (pseudo)-random permutation π
for all i_PE 2^h'/n ≤ j < (i_PE + 1) 2^h'/n do
    P := P_root
    for all 0 ≤ l < h' do
        (P, P') := split(P)
        if π(j) bitand 2^l ≠ 0 then P := P'
    rof
    P := work(P, ∞)                                (* exhaust this subproblem *)
```

Abbildung 8.2: Randomisierte statische Lastverteilung.

der linke bzw. rechte Nachfolger von P falls $(P_0, P_1) = \text{split}(P)$. Werden die Blätter von links nach rechts durchnumeriert, so erzeugt und bearbeitet PE i die Teilprobleme mit den Nummern $\pi(i_{\text{PE}} \frac{2^{h'}}{n})$ bis $\pi((i_{\text{PE}} + 1)\frac{2^{h'}}{n} - 1)$. Auf diese Weise wird jedes Teilproblem genau auf einem PE bearbeitet, ohne daß dazu weitere Kommunikation nötig wäre. Abbildung 8.2 zeigt den Pseudocode für das Verfahren. In Abbildung 8.3 findet sich ein Beispiel für einen möglichen Ablauf des statischen Lastverteilungsalgorithmus.

Der folgende einfache Satz zeigt, daß der Algorithmus eine Variante des in Definition 8.7 beschriebenen „zufälligen Verteilens“ implementiert:

Satz 8.11. *Ist die in Algorithmus 8.2 gewählte Permutation eine echte Zufallspermutation, so wird die Strategie des zufälligen Verteilens realisiert.*

Beweis. Durch Wahl einer Zufallspermutation π über $\mathbb{N}_m$ ($m = 2^{h'}$) wird immer eine Teilproblemzuordnung aus $\bar{\Omega}$ erzeugt, und zwar jede mit der gleichen Wahrscheinlichkeit $\frac{(\frac{m}{n}!)^n}{m!}$: Es gibt $m!$ Permutationen über $\mathbb{N}_m$. Wird ein Element aus $\bar{\Omega}$ festgelegt, so bleibt die Wahl zwischen den $(\frac{m}{n}!)^n$ Realisierungen, die sich nur durch lokale Vertauschungen auf einem der n PEs unterscheiden. □

Leider ist die Erzeugung einer echten Zufallspermutation der Größe $2^{h'}$ recht aufwendig. Die besten bekannten Algorithmen (wie z.B. die in Abschnitt 4.3.1 beschriebenen) benötigen dafür Kommunikation im Umfang von $\Omega(2^{h'})$ Wörtern der Länge h'.[3)] Pseudozufallspermutationen, wie die in Abschnitt 4.3.2, kommen dagegen mit einem Broadcast von konstant vielen Wörtern aus. Die Berechnung des ersten Permutationswertes auf jedem PE kostet höchstens $O(h'^2)$ Wortoperationen und jeder weitere Funktionswert kann mit $O(h')$ Wortoperationen berechnet werden. Die folgenden Aussagen sind unter dem Vorbehalt zu verstehen, daß die gewählte Permutation sich wie eine Zufallspermutation verhält.

[3)] Es wird im folgenden davon ausgegangen, daß $h' \in O(\log n)$. Gemäß der Vereinbarung in Abschnitt 2.2 lassen sich Zahlen der Länge h' deshalb in konstant vielen Maschinenwortoperationen verarbeiten. Die folgende Analyse wird zeigen, daß statische Lastverteilung ohnehin ungeeignet ist, falls schneller wachsende h' benötigt werden.

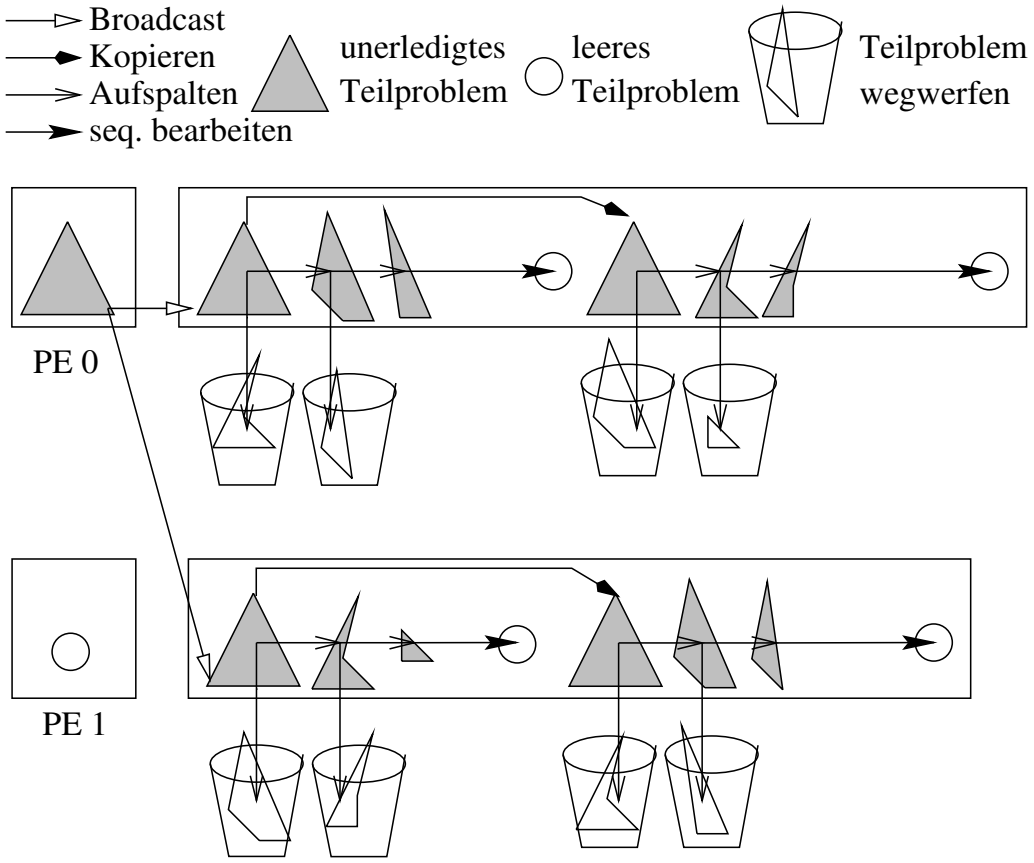

Abbildung 8.3: Möglicher Ablauf des statischen Lastverteilungsalgorithmus für $n = 2$, $h' = 2$ und $\pi = \{0 \to 3, 1 \to 0, 2 \to 2, 3 \to 1\}$. Spaltfunktion und Probleminstanz verhalten sich genauso wie in Abbildung 2.5.

8.3.2 Analyse

Die untere Schranke in Abschnitt 5.3 beruht darauf, daß kein Teilproblem größer $\frac{T_{\text{seq}}}{n}$ sein darf, damit eine gute Lastverteilung ohne weitere Problemaufspaltungen überhaupt möglich ist. Mit Hilfe des abstrakten Modells aus Abschnitt 8.1 kann nun andererseits gezeigt werden, daß selbst statische Lastverteilung effizient ist, wenn Teilprobleme nochmal um einen logarithmischen Faktor kleiner sind.

Satz 8.12. *Für die Ausführungszeit von Algorithmus 8.2 gilt*

$$\forall \varepsilon > 0 : \exists c : \left(\forall P \in \mathcal{P} : \text{gen}(P) \geq h' \to T(P) \leq \frac{T_{\text{seq}}}{cn \log n} \right)$$
$$\to \mathbf{E}T_{\text{par}} \preceq (1+\varepsilon)\frac{T_{\text{seq}}}{n} + \mathrm{O}\left(l + T_{\text{coll}} + h' \frac{2^{h'}}{n} T_{\text{split}} \right)$$

falls die verwendeten Permutationen sich wie Zufallspermutationen verhalten.

Beweis. Die Einheit der Zeitmessung sei o.B.d.A. so festgelegt, daß $T_{\text{seq}} = 1$. Nach Satz 8.10 kann c so gewählt werden, daß die erwartete maximale Last eines PE durch $(1+\varepsilon)\frac{T_{\text{seq}}}{n}$ nach

oben beschränkt ist, wenn kein Teilproblem der Generation $\geq h'$ größer als $\frac{T_{\text{seq}}}{cn\log n}$ ist. Der Broadcast des Wurzelproblems läßt sich in $O(l+T_{\text{coll}})$ Zeitschritten bewältigen. Weiterhin muß jedes PE $\frac{2^{h'}}{n}$ Teilprobleme erzeugen, wofür jeweils h' Spaltoperationen durchzuführen sind. Der Aufwand für die Schleifenkontrolle fällt asymptotisch nicht ins Gewicht. Das gleiche gilt für die Berechnung der Pseudozufallspermutation, wenn die oben beschriebenen Verfahren verwendet werden. □

Leider gibt es baumförmige Berechnungen, für die h' sehr nahe bei h gewählt werden muß, um diese Bedingung zu erfüllen. Immerhin gilt das folgende Pendant zur unteren Schranke 5.7:

Satz 8.13. *Für alle baumförmigen Berechnungen und jedes $\varepsilon > 0$ gibt es ein*

$$h' \in h - \log \frac{T_{\text{seq}}}{T_{\text{atomic}}} + \log n + \log\log n + O(1)\,,$$

so daß das Kriterium aus Satz 8.12 erfüllt ist.

Beweis. Aus der Definition baumförmiger Berechnungen läßt sich leicht folgern, daß es kein Teilproblem P mit $\text{gen}(P) \geq h - \log \frac{T_{\text{seq}}}{T_{\text{atomic}}} + \log n + \log\log n + \log c = h - \log \frac{T_{\text{seq}}}{cn\log n T_{\text{atomic}}}$ und $T(P) \geq \frac{T_{\text{seq}}}{cn\log n}$ geben kann. □

Die schwierig zu balancierenden Probleminstanzen aus Abschnitt 5.3 haben die Eigenschaft, daß Teilprobleme mit kleiner Generation sehr ungleichmäßig aufgespalten werden. Für den Fall, daß mehr über die Spaltoperation bekannt ist, sei deshalb wieder das α-Spaltmodell herangezogen, bei dem *kein* Aufruf von split(P) ein Teilproblem kleiner αP zurückliefern darf. Mit Hilfe von Satz 8.12 läßt sich dann der folgende Satz zeigen:

Satz 8.14. *Gegeben sei eine baumförmige Berechnung mit α-Spalt-Parameter α. Sei $x(\alpha) := \frac{1}{\log \frac{1}{1-\alpha}}$. Sei $\varepsilon > 0$ beliebig. Dann gibt es ein $h' \in x(\alpha)(\log n + \log\log n) + O(1)$, so daß für die Ausführungszeit von Algorithmus 8.2 gilt*

$$\mathbf{E}T_{\text{par}} \preceq (1+\varepsilon)\frac{T_{\text{seq}}}{n} + O\Big(T_{\text{atomic}} + T_{\text{coll}}(l) + T_{\text{split}}(n^{x(\alpha)-1}(\log n)^{x(\alpha)+1})\Big)$$

falls die verwendeten Permutationen sich wie Zufallspermutationen verhalten.

Beweis. Sei $h' = x(\alpha)(\log n + \log\log n) + a$. Beim Spalten eines Teilproblems P kann kein Teilproblem entstehen, das größer als $(1-\alpha)T(P)$ ist. Folglich gibt es nach h'-maligem Spalten des Wurzelproblems kein Teilproblem mehr, das größer als $(1-\alpha)^{h'}T_{\text{seq}} = \cdots = (1-\alpha)^a \frac{T_{\text{seq}}}{n\log n}$ ist. Bei geeigneter Wahl von a ist die Prämisse von Satz 8.12 also erfüllt. Durch Einsetzen von h' in den Term für die sich ergebende Ausführungszeit ergibt sich die Behauptung. □

Die sich ergebenden Formeln sind auf den ersten Blick etwas kompliziert, lassen sich aber relativ leicht interpretieren. Die zu wählende Spalttiefe hängt vor allem von der Güte der Spaltoperation und der Prozessorzahl ab. Abbildung 8.4 zeigt den Verlauf von $x(\alpha)$. Für Werte nahe bei $\frac{1}{2}$ (perfekte Zweiteilung) ergibt sich ein Wert nahe bei 1. Für kleine α steigt

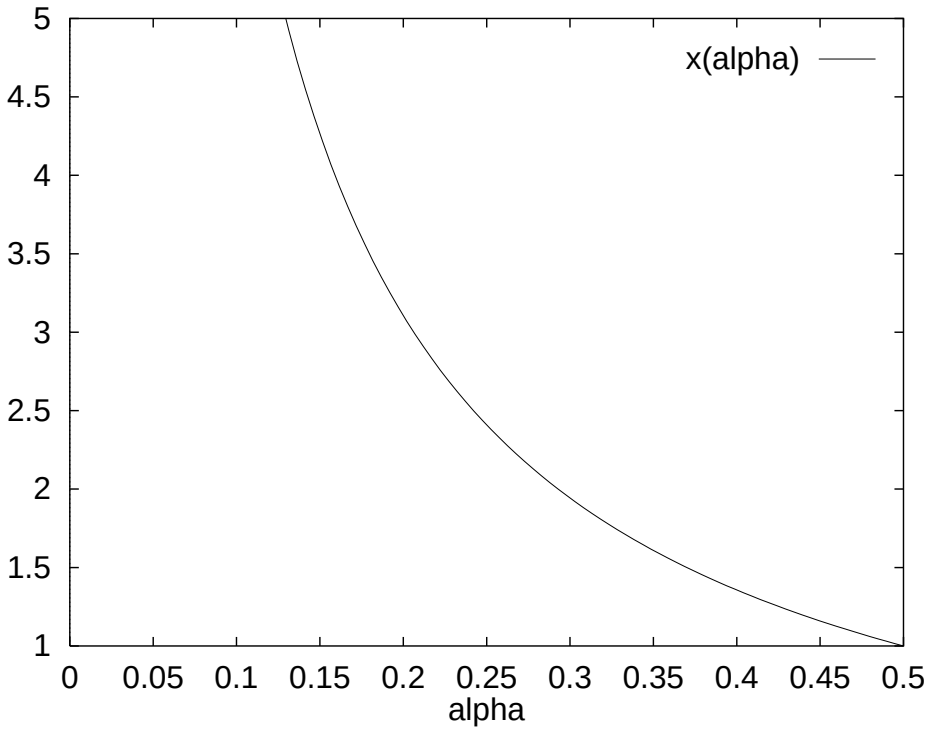

Abbildung 8.4: Entwicklung von $x(\alpha)$.

der Wert aber sehr schnell. Die Abhängigkeit von h' von der Prozessorzahl ist durch einen Faktor $\log n + \log\log n$ gegeben. Der Summand $\log n$ dieses Faktors wird bereits benötigt, um die untere Schranke einzuhalten. Der Summand $\log\log n$ tritt auf, weil die randomisierte statische Lastverteilung erst funktioniert, wenn die maximale Teilproblemgröße um einen weiteren logarithmischen Faktor kleiner gewählt wird. Der zusätzliche Summand a wird benötigt, um den konstanten Faktor c in der maximalen Teilproblemgröße aus Satz 8.12 richtig einzustellen. (Bei genauerer Rechnung zeigt sich, daß $a \in \Omega(\frac{1}{\varepsilon^2})$ zu wählen ist.)

Da h' im Exponenten der parallelen Ausführungszeit auftritt, wirken sich diese Größen stark auf den Aufwand für Problemaufspaltungen aus. Insgesamt muß eine polynomielle Zahl (in n) von Aufspaltungen auf jedem PE vorgenommen werden. Der Exponent hängt dabei ausschließlich von α ab. Bei fast perfekter Aufspaltung ist er nahe Null; dann wird die parallele Ausführungszeit asymptotisch von der sequentiellen Berechnung und dem Zeitbedarf für den Broadcast dominiert. Die Tatsache, daß jedes Teilproblem jeweils neu vom Wurzelproblem abgeleitet wird, sowie der $\log\log n$-Term in h' steuern einen polylogarithmischen Faktor bei.

Interessant sind vor allem Problemklassen mit α nahe bei $\frac{1}{2}$. Ist das Verbindungsnetzwerk z.B. ein r-dimensionales Gitter und ist $\alpha < 1 - 2^{-\frac{r}{r+1}}$, so ist der Exponent $x(\alpha) - 1$ kleiner $\frac{1}{r}$, und folglich kann der Aufwand für Problemaufspaltung für große n vom Kommunikationsaufwand dominiert werden. (Für $r = 1$ ergibt sich die Bedingung $\alpha > 0.29$ und für $r = 2$ ist es $\alpha > 0.37$.) Da der Kommunikationsaufwand statischer Lastverteilung aber an der trivialen unteren Schranke $\Omega(d + l)$ liegt, ergibt sich ein asymptotisch optimales Verfahren.

Trotzdem sind die hier und in Abschnitt 5.3 erzielten Ergebnisse in erster Linie ein Negativergebnis. Während im Modell von KRUSKAL UND WEISS (1985) $O(\log n)$ Teilprobleme pro PE für guten Lastausgleich genügen, müssen zumindest hier polynomiell viele Teilproble-

me pro PE erzeugt werden. Bei ungenauer Spaltoperation ist der auftretende Exponent dazu sehr groß. Schlimmer noch, es läßt sich leicht nachrechnen, daß es für $h \notin \mathrm{O}\left(\log \frac{T_{\mathrm{seq}}}{T_{\mathrm{atomic}}}\right)$ kein konstantes $\alpha > 0$ geben kann; dann ist die Anzahl zu erzeugender Teilprobleme noch nicht einmal polynomiell.

8.4 Kombination mit dynamischer Lastverteilung

Randomisierte statische Lastverteilung als alleiniger Lastverteiler ist nach den Ergebnissen der letzten Abschnitte offenbar nur in Spezialfällen sinnvoll. Es gibt aber eine Anzahl von Kombinationsmöglichkeiten mit dynamischer Lastverteilung, die den geringen Kommunikationsaufwand statischer Verfahren mit der Robustheit dynamischer Verfahren vereinen. Begonnen wird in Abschnitt 8.4.1 mit der naheliegenden Idee, jedem PE zu Beginn ein einziges zufälliges Teilproblem zuzuteilen und erst dann einen dynamischen Lastverteiler zu starten. In Abschnitt 8.4.2 stellt sich heraus, daß bei Kombination mit Varianten von zufälligem Anfragen in vielen Fällen eine asymptotische Verbesserung gegenüber dem Grundalgorithmus erreicht werden kann.

8.4.1 Randomisierte zufällige Initialisierung

In Abschnitt 2.14 wurde ein generischer Algorithmus für empfängerveranlaßte dynamische Lastverteilungsverfahren eingeführt. Bewußt wurde dort auf eine genauere Spezifikation der Initialisierung verzichtet. Die einfachste und schnellste Initialisierung ist sicherlich die, einem PE (z.B. Nr. 0) das Wurzelproblem zuzuteilen und die Versorgung der anderen PEs dem dynamischen Lastverteilungsverfahren zu überlassen. Allerdings ist die Lastverteilung dann während einer Anfangsphase sehr schlecht.

Dies läßt sich verbessern, indem ein Initialisierungsverfahren gewählt wird, das zunächst jedem PE ein Teilproblem zuteilt. Dafür kann Algorithmus 8.2 verwendet werden. Indem $h' = \log n$ gesetzt wird, erhält jedes PE genau ein Teilproblem. (Eine Verallgemeinerung für den Fall, daß n keine Zweierpotenz ist, ist recht einfach.)

Ein sehr ähnliches Verfahren findet sich bereits in EL-DESSOUKI UND HUEN (1980) wird dort allerdings als alleiniger Lastverteiler eingesetzt. In REINEFELD (1994) wird ein Initialisierungsverfahren verwendet, das mit wenig Kommunikation auskommt, aber zur Generierung der Teilprobleme wird Breitensuche eingesetzt, die einen recht hohen lokalen Rechenaufwand von $\Omega(n)$ aufweist[4)] und außerdem einen hohen Speicheraufwand hat.

Ein wichtiger Unterschied zu vorher verwendeten Verfahren besteht darin, daß nun die Teilprobleme zufällig plaziert werden. Dadurch kann man hoffen, daß besonders große Teilprobleme nicht zu dicht beieinander liegen. Dadurch erscheinen Lastverteilungsprobleme mit Nachbarschaftskommunikation, wie sie in RAO UND KUMAR (1987b) analysiert werden, plötzlich wieder interessanter. Die dort gemachte Aussage, daß Nachbarschaftsommunikation zu „Klumpenbildung“ führt (siehe auch Abschnitt 2.4.2) geht nämlich davon aus, daß anfangs nur ein PE Arbeit hat. Leider läßt sich leicht zeigen, daß reine Nachbarschaftskommunikation auf den meisten Netzwerken trotzdem wenig aussichtsreich ist. Die Grundidee, nämlich globale Kommunikation durch zufällige Initialisierung unnötig zu machen, wird jedoch im folgenden Abschnitt weiterverfolgt.

[4)] Im Rahmen des Problemmodells „baumförmige Berechnung“ ist der Zeitaufwand ziemlich genau nT_{split}. In den meisten realen Anwendungen läßt sich eine Breitensuche aber etwas schneller realisieren.

8.4.2 Kombination mit zufälligem Anfragen

Wird zufälliges Anfragen mit dem oben beschriebenen Initialisierungsverfahren kombiniert, so wird dadurch die in Abschnitt 6.3 untersuchte Einschwingphase eingespart, die im Mittel $\log n + \log\log n + O(1)$ Spaltoperationen und globale Teilproblemübertragungen benötigt. Ein Broadcast ist zwar i. allg. schneller, aber auf die Gesamtausführungszeit dürfte dies nur eine deutliche Auswirkung haben, wenn h nahe bei $\log n$ liegt und relativ kleine Probleme vorliegen, für die die parallele Gesamtausführungszeit in der Größenordnung der Initialisierung liegt.

Bei naiver Kombination mit zufälligem Anfragen wird nicht ausgenutzt, daß die Initialisierung die Daten bereits zufällig verteilt. Da für Lastanfragen ein global zufälliges PE ausgewählt wird, spielt die räumliche Verteilung der Daten keine Rolle. Zumindest für das α-Spaltmodell läßt sich dagegen zeigen, daß eine besonders einfache Variante des in Abschnitt 7.5 beschriebenen Algorithmus für lokales zufälliges Anfragen von der zufälligen Verteilung profitiert:

- Initialisiere jedes PE mit einem zufälligem Teilproblem.
- Teile das Netzwerk in würfelförmige Teilgitter mit Seitenlänge $n^{1/r}/a$.
- Führe lokal in jedem Teilgitter zufälliges Anfragen aus.

Satz 8.15. *Gegeben sei eine baumförmige Berechnung mit α-Spalt-Parameter α. Sei $\varepsilon > 0$ beliebig. Sei a eine Funktion, die höchstens polylogarithmisch in n wächst. Sei außerdem $\frac{T_{\text{seq}}}{T_{\text{atomic}}}$ polynomiell in n.*[5] *Dann gilt für die parallele Ausführungszeit von lokalem zufälligem Anfragen mit Initialisierung:*

$$\mathbf{E}T_{\text{par}} \preceq (1+\varepsilon)\frac{T_{\text{seq}}}{n} + O\left(T_{\text{atomic}} + l + n^{1/r} + \frac{n^{1/r} l \log n}{a} + T_{\text{split}} \log n\right)$$

für hinreichend großes n.

Die Grundidee für den Beweis besteht darin, die Analyse von zufälligem Anfragen auf ein einzelnes Teilgitter anzuwenden und die Analyse randomisierter statischer Lastverteilung heranzuziehen, indem ein Teilgitter als ein einziger "Metaprozessor" aufgefaßt wird. Die Teilergebnisse lassen sich mit Hilfe der Rechenregeln aus Kapitel 3 zusammensetzen.

Beweis. Die Initialisierung ist in Zeit $O\left(n^{1/r} + l + T_{\text{split}} \log n\right)$ möglich. Sei $T_{\max}$ die maximale sequentielle Gesamtarbeit, die einem Teilgitter zugeteilt wird. Dann läßt sich die für lokales zufälliges Anfragen benötigte Ausführungszeit in einem einzelnen Teilgitter nach Satz 6.1 durch

$$\sqrt{1+\varepsilon}\frac{T_{\max} a^r}{n} + \tilde{O}\left(T_{\text{atomic}} + h\left(\frac{n^{1/r} l}{a} + T_{\text{split}}\right)\right)$$

[5] Diese Annahme dient der Vereinfachung der Darstellung. Andernfalls geht es um sehr große Probleme, deren effiziente Parallelisierung ohnehin nicht schwierig ist.

abschätzen. Wie schon in Abschnitt 2.3.5 diskutiert, ist für konstantes α immer $h \in \Theta\left(\log \frac{T_{\text{seq}}}{T_{\text{atomic}}}\right)$. Da außerdem $\frac{T_{\text{seq}}}{T_{\text{atomic}}}$ polynomiell in n ist, folgt $h \in \Theta(\log n)$ und die für zufälliges Anfragen benötigte Zeit ist

$$\sqrt{1+\varepsilon}\frac{T_{\max}a^r}{n} + \tilde{O}\left(T_{\text{atomic}} + \frac{n^{1/r}l\log n}{a} + T_{\text{split}}\log n\right) \quad .$$

Nach der Maximumregel aus Satz 3.11 gilt die gleiche Aussage auch für das Teilgitter, in dem das zufällige Anfragen am langsamsten ist. Es genügt nun zu zeigen, daß $\mathbf{E}T_{\max} \leq \sqrt{1+\varepsilon}\frac{T_{\text{seq}}}{a^r}$. Hierzu wird Satz 8.10 für den Fall angewendet, daß a^r Prozessoren je n/a^r Teilprobleme zugeteilt werden. Es gilt zu überprüfen, daß die maximale Teilproblemgröße sich für beliebige Konstanten c durch $\frac{T_{\text{seq}}}{ca^r \ln a^r}$ nach oben abschätzen läßt. Für hinreichend große n ist diese Bedingung erfüllt, da der Nenner der Teilproblemgrößenschranke $(1-\alpha)^{\log n} = n^{-\log \frac{1}{1-\alpha}}$ polynomiell in n wächst während $ca^r \ln a^r$ polylogarithmisch in n ist. □

Falls l nicht zu schnell wächst, läßt sich mit lokalem zufälligem Anfragen sogar die untere Schranke aus Satz 5.2 erreichen. Durch Wahl von $a \in \Omega(l\log n)$ wird der Kommunikationsaufwand der lokalen Lastverteilung nämlich soweit gedrückt, daß er gegenüber dem Aufwand für die Initialisierung nicht mehr ins Gewicht fällt.

Korollar 8.16. *Ist l polylogarithmisch in $\frac{T_{\text{seq}}}{T_{\text{atomic}}}$ (und damit in n) so läßt sich in Satz 8.15 a so wählen, daß*

$$\mathbf{E}T_{\text{par}} \preceq (1+\varepsilon)\frac{T_{\text{seq}}}{n} + O\left(T_{\text{atomic}} + l + n^{1/r} + T_{\text{split}}\log n\right)$$

für hinreichend großes n.

Leider ist die konkrete Wahl von a nicht einfach. Deshalb ist es sinnvoll, einige der in Abschnitt 7.5 beschriebenen Ergänzungen zu verwenden, um auch bei falscher Wahl von a eine gute Lastverteilung zu garantieren. Dann entsteht ein Algorithmus, der die drei Lastverteilungsansätze aus den Kapiteln 6, 7 und 8 vorteilhaft in sich vereinigt. Der Initialisierung ersetzt dabei den ersten Zyklus des Fragen-und-Mischen-Algorithmus. Außerdem zeigt sich, daß zumindest für das α-Spaltmodell kaum globale Mischoperationen nötig sind. Dies rechtfertigt die in Abschnitt 7.4 beschriebenen Maßnahmen zur Einsparung von Mischoperationen.

8.5 Zusammenfassung

Ein abstraktes Modell

Im Gegensatz zu früher behandelten Modellen für randomisierte statische Lastverteilung wird hier nicht die Anzahl Teilprobleme sondern die maximale Teilproblemgröße in den Mittelpunkt gestellt. Dadurch wird es möglich, eine Analyse für das mittlere Verhalten von Lastverteilern im schlimmsten Fall zu machen, ohne zusätzliche, schwer zu rechtfertigende Annahmen über die Herkunft der Teilprobleme machen zu müssen. Die hier behandelten einfachen randomisierten Lastverteilungsalgorithmen *unabhängiges Zuordnen* und *zufälliges Verteilen* sind in der Lage, eine gute Lastverteilung zu erreichen, wenn die maximale Teilproblemgröße

um einen Faktor $O(\log n)$ kleiner ist als die Last pro PE. Dabei ist zufälliges Verteilen mindestens so gut wie unabhängiges Zuordnen. Da hier die Anwendung des abstrakten Modells auf baumförmige Berechnungen im Vordergrund steht, wird eine weitergehende spieltheoretische Untersuchung des Modells nicht durchgeführt. So läßt sich erwarten, daß die einfachen randomisierten Lastverteilungsstrategien in gewissem Sinne optimal sind, und zwar unabhängig von der verwendeten Strategie des Gegners. Umgekehrt dürften Max-Strategien für den Gegner optimal sein und zwar nicht nur gegen unabhängiges Zuordnen sondern gegen alle Lastverteilungsstrategien. Vermutlich ist es für den Gegner am besten, die nichtleeren Teilprobleme zufällig festzulegen.

Randomisierte statische Lastverteilung

Wird das abstrakte Modell auf baumförmige Berechnungen angewandt, so lassen sich daraus hinreichende Anforderungen für eine gute Lastverteilung herleiten. Die sich ergebende maximale Teilproblemgröße ist nur einen logarithmischen Faktor kleiner als die triviale notwendige Bedingung, daß kein Teilproblem deutlich mehr als den $\frac{1}{n}$-ten Teil der Gesamtproblemgröße ausmachen darf. Trotzdem ist statische Lastverteilung für baumförmige Berechnungen mit nicht gleichmäßig wirksamer Spaltoperation im allgemeinen ungeeignet. Günstiger sieht es für „gutmütige“ Probleme aus, die z.B. dem α-Spaltmodell genügen. Für α nahe dem bestmöglichen Wert $\frac{1}{2}$ hält sich der Aufwand für das Erzeugen hinreichend vieler Teilprobleme in vernünftigen Grenzen. Dann kann statische Lastverteilung durchaus schneller sein als dynamische Algorithmen wie zufälliges Anfragen oder Fragen-und-Mischen. Für Gitter ergibt sich u.U. sogar ein asymptotisch optimaler Algorithmus.

Interessant kann statische Lastverteilung auch dann sein, wenn es weniger um eine Effizienz nahe 1, sondern um eine möglichst große Geschwindigkeit geht. Ist eine baumförmige Berechnung zu klein, um die zur Verfügung stehende Anzahl PEs *effizient* auszunutzen, so macht es oft trotzdem Sinn, so viele PEs einzusetzen wie möglich. (Solange die Ausführungszeit dadurch nicht vergrößert wird.) Statische Lastverteilung ist dann vorteilhaft, weil sie mit minimaler Kommunikation auskommt. Sehr kleine Teilprobleme treten z.B. bei Anwendungen auf, die auf iterativem Vertiefen beruhen (siehe auch Abschnitt 2.3.3). In den ersten Iterationen ist der Suchbaum noch so klein, daß eine effiziente Parallelisierung nicht möglich ist. Hinzu kommt, daß die Problembeschreibung in jeder Iteration die gleiche ist und es meist keine Ergebnisse einzusammeln gibt. (Siehe auch Abschnitt A.4.4.) In diesem Fall reduziert sich der Kommunikationsaufwand auf eine einzige globale Synchronisation. Auf Maschinen, die Synchronisation durch Hardware unterstützen (z.B. Cray T3D/T3E) ist der Zeitaufwand dafür vernachlässigbar[6)] – es können also alle zur Verfügung stehenden PEs nutzbringend eingesetzt werden. Einige experimentelle Erfahrungen mit diesem Szenario sind in Abschnitt A.7 beschrieben.

Anwendungen für paralleles Theorembeweisen

Beim parallelen Theorembeweisen und verwandten Problemklassen ist der Suchraum oft so groß (manchmal unendlich groß), daß er nicht komplett durchsucht werden kann. Andererseits gibt es mit etwas

[6)] $\log n$ Gatterlaufzeiten plus $\frac{\text{physikalischer Maschinendurchmesser}}{\text{Signalausbreitungsgeschwindigkeit}}$.

Glück sehr viele Lösungen, und es genügt, eine davon zu finden. Sind keine starken Heuristiken bekannt, die die Suche leiten können, ist es dann oft günstig, verschiedene Prozessoren in verschiedenen zufälligen Teilbäumen suchen zu lassen (JANAKIRAM ET AL., 1987; NATARAJAN, 1989; ERTEL, 1992). Bei „ungleichmäßiger“ Verteilung der Lösungen ergibt sich dadurch im Mittel eine Effizienz größer eins! Bisherige Ansätze beruhen darauf, auf den einzelnen PEs eine sequentielle Suche am gleichen Problem durchzuführen, aber die Nachfolger eines Suchbaumknotens zufällig anzuordnen. Ist der Suchraum aber kleiner als erwartet, müssen sich die von den einzelnen PEs bearbeiteten Teilbäume zwangsläufig überlappen. Gibt es überhaupt keine Lösung, durchsuchen alle PEs den gesamten Baum. Die *Beschleunigung* liegt dann zwangsläufig unter eins. Da bei Theorembeweisen oft iteratives Vertiefen eingesetzt wird, bedeutet dies, daß die ersten Iterationen überhaupt nicht beschleunigt werden. Die in diesem Kapitel beschriebene zufällige Baumzerlegung bietet sich deshalb als Verbesserung an. Der Suchbaum wird auf zufällige Weise in disjunkte Teilbäume zerlegt. In diesen kann dann wieder zufällig sequentiell gesucht werden. Da Überlappungen nun ausgeschlossen sind, kann sich die Ausführungszeit nur verringern. Werden mehrere Teilprobleme pro PE erzeugt, mag es hier aber ratsam sein, diese nicht nacheinander, sondern im Zeitscheibenverfahren gleichzeitig abzuarbeiten. Wird ein PE arbeitslos, spricht außerdem nichts dagegen, dynamische Lastverteilung einzusetzen.

Kombination mit dynamischer Lastverteilung

Die Initialisierung jedes PE mit einem Teilproblem durch statische Baumzerlegung ist so einfach und effizient, daß wohl jedes dynamische Lastverteilungsverfahren davon profitieren kann. Bei Probleminstanzen, bei denen die ersten $\log n$ Aufspaltungen eine sehr ungleichmäßige Zerlegung bewirken, ist der erzielbare Gewinn aber begrenzt. Bei gutmütigeren Problemen, die z.B. dem α-Spaltmodell genügen, ist dagegen eine asymptotisch signifikante Verbesserung erreichbar. Dafür ist es entscheidend, daß das dynamische Verfahren die zufällige Verteilung der Teilprobleme ausnutzt. So läßt sich der Kommunikationsaufwand für zufälliges Anfragen auf Gittern um einen Faktor bis zu $O(l \log n)$ reduzieren, indem der Kommunikationsaufwand für Lastanfragen eingeschränkt wird.

Ergebnisse

Das hier betrachtete abstrakte Modell für statische Lastverteilung von Teilproblemen unbekannter Größe ist offenbar das erste, das genaue Aussagen für den schlechtesten Fall zuläßt und keine Annahmen über Verteilungen von Teilproblemgrößen machen muß, die sich u.U. schwer rechtfertigen lassen.

Die Idee, ausgehend vom Wurzelproblem ohne Kommunikation disjunkte Teilprobleme zu erzeugen ist nicht neu. Neu ist aber, daß die Teilprobleme dabei zufällig plaziert werden können. Der Ansatz, statische Lastverteilung zu verbessern, indem jedem PE viele Teilprobleme zugeordnet werden, ist ebenfalls bekannt. Aber erst durch die Kombination mit dem Verfahren zur schnellen Erzeugung zufällig plazierter disjunkter Teilprobleme wird dieser Ansatz auch für baumförmige Berechnungen sinnvoll anwendbar.

Die Kombination schneller Initialisierungsverfahren mit dynamischer Lastverteilung wurde bisher rein empirisch betrachtet. Die in Abschnitt 8.4 analysierten Verfahren sind deshalb ebenfalls Neuland. Die Kombination mit lokalem zufälligem Anfragen ist offenbar das erste Verfahren, das wenigstens auf Gitterrechnern für Probleme im α-Spaltmodell nachweisbar asymptotisch optimal ist – es erreicht die Durchmesserschranke aus Abschnitt 5.2. Die reine statische Lastverteilung erreicht diese Schranke immerhin für α nahe bei $\frac{1}{2}$.

Kapitel 9

Schluß

In Abschnitt 9.1 werden die wichtigsten Ergebnisse dieser Arbeit zusammengefaßt, und in Abschnitt 9.2 werden sich anschließende zusätzliche Fragestellungen diskutiert.

9.1 Was ist erreicht?

Baumförmige Berechnungen sind eine nützliche Abstraktion von vielen realen Anwendungen, vor allem solchen, die auf Tiefensuche in großen implizit gegebenen Bäumen beruhen. In vielen Fällen werden Anwendungen vollständig oder doch in ihren essentiellen Teilen modelliert. In anderen Fällen sind baumfömige Berechnungen immerhin geeignet, den Lastverteilungsaspekt des Problems abzubilden.

Für all diese Anwendungen steht nun ein Spektrum an Lastverteilungsalgorithmen zusammen mit einer detaillierten Analyse für viele Klassen von Parallelrechnerarchitekturen zur Verfügung. Zufälliges Anfragen hat sich trotz seiner extremen Einfachheit als beweisbar effizient herausgestellt, und falls globale Kommunikation nicht deutlich teurer als lokale Kommunikation ist, ist es asymptotisch kaum noch verbesserbar. Auf vielen Netzwerken läßt sich durch den Fragen-und-Mischen-Algorithmus nochmals eine Verbesserung erzielen, indem der Kommunikationsaufwand um einen logarithmischen Faktor verringert wird. Weitere Einsparung von Kommunikation läßt sich durch randomisierte statische Verteilung der Teilprobleme erreichen, insbesondere dann, wenn die Problemaufspaltungsfunktion gleichmäßig wirksam ist. All diese Ansätze lassen sich außerdem vielfältig miteinander kombinieren.

Für eine konkrete Einzelanwendung ist die Verbesserung des Lastverteilungsalgorithmus nur eine von vielen „Schrauben“ an denen gedreht werden kann. Dort wird man sich oft mit einfachen Varianten von (z.B.) zufälligem Anfragen zufrieden geben. Die Algorithmen lassen sich aber auch anwendungsunabhängig in einer Bibliothek wie PIGSeL oder im Laufzeitsystem einer Programmiersprache wie parallelem Prolog realisieren. Dort kommt die Implementierung einer Vielzahl von Anwendungen zugute. Gerade hier sind die beweisbaren Laufzeitschranken ein wichtiges Qualitätsmerkmal, weil sie verläßliche Voraussagen über den Erfolg der Parallelisierung ermöglichen.

9.2 Welche Fragen schließen sich an?

Wie in der Wissenschaft üblich, wirft auch diese Arbeit mehr neue Fragen auf als sie beantwortet. Deutlich wird dies z.B. bei der Implementierung. Während die Unzufriedenheit mit dem Kenntnisstand bei der Analyse existierender Algorithmen ein Ausgangspunkt dieser Arbeit war, haben sich aus der theoretischen Beschäftigung mit dem Thema neue, asymptotisch bessere Algorithmen ergeben, deren praktische Bewertung und Weiterentwicklung ein weites und interessantes Feld ist. Zum Beispiel: Welche der Varianten von Fragen-und-Mischen erzielt auf heutigen Parallelrechnern eine bessere Leistung als zufälliges Anfragen? Oder wie können Spaltfunktionen verbessert werden, damit mehr Anwendungen von zufälliger statischer Lastverteilung (als Initialisierung) profitieren können?

Sehr interessant ist die Frage, wie Modelle aussehen müssen, die eine größere Klasse von Anwendungen (genauer) modellieren, und wie die hier erzielten Ergebnisse sich übertragen lassen. Zum Beispiel könnten asynchrones zufälliges Anfragen und Fragen-und-Mischen für die in BLUMOFE UND LEISERSON (1994) betrachteten vielfädigen Berechnungen angewendet werden. Oder, es könnte der Begriff der Wichtigkeit oder Priorität von Teilproblemen eingeführt werden, um damit Anwendungen wie Branch-and-bound oder Spielbaumsuche näher zu kommen, bei denen die Größe des Suchbaums von der Art seiner Traversierung abhängt.

Der Erfolg sehr einfacher Zufallspermutationen (nämlich Verschiebungen) bei synchronem zufälligen Anfragen läßt vermuten, daß ähnliche Techniken auch bei Fragen-und-Mischen oder zufälligem Verteilen *beweisbar* nützlich sein könnten.

Von großem theoretischen Interesse wäre die Entwicklung genauerer unterer Schranken. Kann man zeigen, daß globale oder „fast“ globale Kommunikation sich nur auf Kosten der Anzahl Teilproblemtransporte vermeiden läßt? Können randomisierte Algorithmen untere Schranken für deterministische Algorithmen unterbieten? Stellen sich weitere Algorithmus/Maschine/Anwendungskombinationen als asymptotisch optimal heraus?

Eine wichtige Architekturklasse, für die Fragen-und-Mischen keine Verbesserung bringt, sind Busnetzwerke. Dort könnte aber ausgenutzt werden, daß ein Broadcast in konstanter Zeit möglich ist. Wenn jedes PE in der Lage ist, mehrere Teilprobleme zu speichern, könnte eine Lastanfrage ein großes Teilproblem auswählen und es an *viele* PEs übermitteln (mindestens n^{δ} mit δ in $(0,1]$), die das Teilproblem – wie bei der statischen Lastverteilung – ohne weitere Kommunikation in individuelle Stücke zerlegen. Um eine Explosion der Anzahl Teilprobleme zu vermeiden, könnten Techniken wie in Abschnitt 4.2 verwendet werden: Arbeitslose PEs schicken nur mit einer bestimmten Wahrscheinlichkeit eine Lastanfrage. Da nun jede Anfrage die Generation des von ihm erreichten Teilproblems gleich um $\Theta(\log n)$ erhöht, könnte es sich herausstellen, daß die Anzahl nötiger Lastanfragen um einen Faktor $\Theta(\log n)$ sinkt.

Anhang A

Implementierung

Theoretische Ergebnisse und Analysen sind oft von geringem praktischen Nutzen, wenn sie nicht durch entsprechende Implementierungen und Messungen begleitet werden. Trotzdem legt diese Arbeit ihren Schwerpunkt eindeutig auf Entwurf und Analyse. Das hat mehrere Gründe. Zum einen gibt es eine Vielzahl praktischer Untersuchungen zur parallelen Tiefensuche, aber ein Defizit an theoretischer Untermauerung. Das Ergebnis ist ein Überfluß schwer vergleichbarer Algorithmen, bei denen kaum nachzuvollziehen ist, welche Eigenschaften nützlich und allgemein verwendbar sind. Zum Beispiel war die Diskrepanz zwischen der beobachteten hohen Effizienz von zufälligem Anfragen und dem schlechten Abschneiden in der angenäherten Analyse in KUMAR UND ANANTH (1991) ein Anlaß, diesen Algorithmus genauer zu analysieren. Die Beobachtung, daß globale Kommunikation auf Netzwerken mit großem Durchmesser teurer ist als lokale Kommunikation, war Auslöser für die Untersuchung von empfängerveranlaßten Lastverteilungsalgorithmen, die globale Kommunikation vermeiden und hat zum Fragen-und-Mischen Algorithmus geführt. Das schlechte Abschneiden statischer Lastverteilung einerseits und ihr verlockendes Potential zur Einsparung von Kommunikation andererseits war Anlaß zu den Untersuchungen in Kapitel 8.

Viele der theoretischen Ergebnisse in dieser Arbeit sind außerdem von einer Art, die einer praktischen Überprüfung nicht bedürfen. Beim Problemmodell wird darauf geachtet, daß Probleminstanzen durch aus der Praxis leicht ableitbare Parameter wie Suchbaumtiefe oder Nachrichtenlänge gekennzeichnet werden. Die Untersuchungen werden für den schlechtesten Fall gemacht, und vereinfachende Annahmen gibt es nur dort, wo praktische Erfahrung dies zulässig erscheinen läßt. (Zum Beispiel Annahmen über das Verhalten von Datentransportalgorithmen, siehe auch Abschnitt 2.2.2).

Trotzdem stellt sich natürlich eine Vielzahl von Fragen, die man gerne anhand einer realen Implementierung beantworten möchte. Nur wenigen Fragen kann aber tatsächlich nachgegangen werden. Die erforderlichen Programme sollten sich mit vertretbarem Aufwand auf heutigen Maschinen in einigermaßen portabler Weise implementieren lassen. Einige interessante Fragen bleiben da leider auf der Strecke. So ist der Grundalgorithmus für Fragen-und-Mischen auf heutigen Maschinen wenig realistisch, weil die Synchronisationen zu teuer werden. Die verfeinerten Versionen, die weniger Synchronisationen benötigen, sind aber recht komplex und würden für eine sinnvolle Implementierung einen effiziente vielfädige Ausfüh-

rung des Programms erfordern. Solche Programmierumgebungen sind aber im Moment noch kaum verfügbar.

Die folgende Themenauswahl wurde getroffen: In Abschnitt A.1 wird der grundsätzliche Aufbau der Bibliothek PIGSeL beschrieben, die als Umgebung für die portable Implementierung der meisten in Abschnitt A.2 beschriebenen Beispielanwendungen verwendet wurde. Abschnitt A.3 gibt einen Überblick über die verwendeten Maschinen. Damit ist die Grundlage geschaffen, um in Abschnitt A.4 Details der Parallelisierung realer Anwendungen auf realen Maschinen darzulegen. Abschnitt A.5 geht der Frage nach, wie die Anwendungen sich durch baumförmige Berechnungen modellieren lassen. In Abschnitt A.6 werden Implementierungsergebnisse für zufälliges Anfragen vorgestellt. Abschnitt A.7 zeigt Stärken und Schwächen von statischer Lastverteilung und Initialisierungsverfahren. Schließlich wird in Abschnitt A.8 mit einer Variante von lokalem zufälligen Anfragen ein Algorithmus betrachtet, der positive Eigenschaften von zufälligem Anfragen, Fragen-und-Mischen und statischer Initialisierung in sich vereint. Abschnitt A.9 gibt eine Zusammenfassung.

A.1 Softwaretechnische Aspekte

Ein großer Teil der Experimente wurde mit Hilfe der wiederverwendbaren Bibliothek PIGSeL (Parallel Implicit Graph Search Library) gemacht die in SANDERS (1994e, 1996d) sowie DIEFENBACH UND SANDERS (1995) beschrieben ist. Dadurch wird es möglich, Anwendungen, Lastverteilungsverfahren und Maschinen frei miteinander zu kombinieren. Abbildung A.1 zeigt die Struktur der Bibliothek.

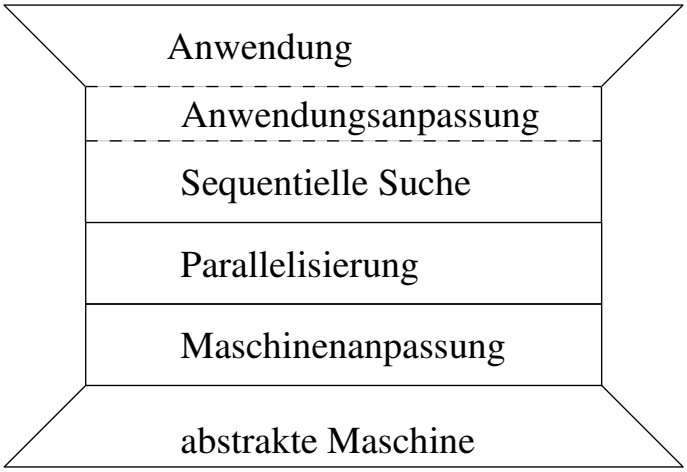

Abbildung A.1: Schichtenstruktur der Bibliothek PIGSeL

Aufbauend auf einer i. allg. nicht portablen abstrakten Maschine (Hardware und Programmierumgebung) sorgt eine Maschinenanpassungsschicht dafür, daß ein minimaler Satz an Nachrichtenaustauschroutinen zur Verfügung steht, der auf allen unterstützten Maschinen funktioniert. Die Maschinenschnittstelle enthält außerdem ein Modul, das kollektive Operationen wie Broadcast und Reduktion zur Verfügung stellt. Dieses Modul ist selbst portabel, kann aber entsprechend optimierte Funktionen der abstrakten Maschine nutzen, wenn diese zur Verfügung stehen.

Die darüberliegende Schicht ist für die Parallelisierung zuständig und nutzt nur die Routinen der Kommunikationsschnittstelle, um Portabilität zu gewährleisten. Neben der Lastverteilung ist diese Schicht auch für die Abwicklung zusätzlicher Protokolle wie dem Einsammeln von Lösungen zuständig.

Kern der Schnittstelle der Lastverteilung nach oben sind baumförmige Berechnungen. Die darüber liegenden Schichten sind dafür zuständig, einen abstrakten Datentyp „Teilproblem“ zu implementieren, der die Operationen „work“ und „split“ unterstützt. Dies kann entweder direkt durch die Anwendung, oder mit Hilfe zweier weiterer Schichten geschehen. Die Schicht für sequentielle Suche implementiert einen generischen sequentiellen Suchalgorithmus[1)] sowie die Funktion zur Aufspaltung des Suchbaums. Die Anwendung muß dann lediglich noch abstrakte Datentypen wie *Suchbaumknoten* und *Lösung* implementieren, die es dem generischen Suchalgorithmus ermöglichen, den Suchbaum zu durchlaufen. Die Anwendungsschnittstelle dient dazu, die Schnittstelle zwischen generischem Suchalgorithmus und Anwendung zu vereinfachen.[2)]

A.2 Betrachtete Anwendungen

Im Rahmen von PIGSeL wurden Golomblineale, das 0/1-Rucksackproblem und das 15-Puzzle implementiert, die in Abschnitten A.2.1–A.2.3 beschrieben sind. (Außerdem gibt es ein Modul für synthetische Suchbäume, das im folgenden aber nicht näher betrachtet wird.)

Zusätzlich standen die Anwendungen „Firing Squad Synchronisation Problem“ und „Spielbaumsuche in synthetischen Bäumen“ für Experimente und Vergleiche zur Verfügung, die auf dem SIMD-Rechner MasPar MP-1 implementiert sind. Diese Anwendungen sind in Abschnitt A.2.4 und A.2.5 beschrieben.

A.2.1 Golomblineale

Ein Golomblineal (BLOOM UND GOLOMB, 1977) der Länge m mit k Markierungen ist ein Lineal mit Markierungen an den ganzzahligen Stellen $m_1, \ldots, m_k$ mit $0 = m_1 < m_2 < \cdots < m_k = m$ und der Eigenschaft $\left| \left\{ m_j - m_i : 1 \leq i < j \leq k \right\} \right| = k(k-1)/2$. Das heißt, mit einem Golomblineal läßt sich eine maximale Zahl Strecken ganzzahliger Länge messen. Wird das Problem als Entscheidungsproblem formuliert, so lautet die Frage an den Suchalgorithmus, ob es zu gegebenen k und m ein Golomblineal mit Länge $\leq m$ gibt.

Es gibt analytische und empirische obere und untere Schranken für das optimale m, die zeigen, daß der interessante Bereich für m in einem recht engen Bereich liegt. Hier ist nur interessant, daß m immer in $\Theta\left(k^2\right)$ ist. Die Implementierung ist zusätzlich in der Lage, das kleinste m zu finden, für das ein Golomblineal existiert, sowie ein solches (optimales) Golomblineal zu bestimmen. Abbildung A.2 zeigt ein optimales Lineal für den Fall $k = 6$, $m = 17$.

[1)] Bisher wurden nur zwei Varianten von Tiefensuche implementiert. Eine Bestensuche ließe sich aber ebenfalls realisieren, ohne daß Anwendungen umgeschrieben werden müßten.

[2)] Würde PIGSeL nur baumförmige Berechnungen unterstützen, wäre diese Schicht verzichtbar, weil die Schnittstelle extrem einfach wäre. Für eine effiziente und allgemeine Behandlung von Protokollen zur Behandlung von Lösungen und zur Beschneidung des Suchbaums ist aber eine relativ große Zahl zusätzlicher Funktionen nötig. Diese flexible aber komplexe Schnittstelle läßt sich durch Spezialisierung auf verschiedene Klassen von Anwendungen wieder vereinfachen.

Abbildung A.2: Golomblineal der Länge 17 mit 6 Markierungen.

Für Golomblineale gibt es Anwendungen in der Interferometrie und im Entwurf effizienter Codes. Hier wurde das Problem aus softwaretechnischen Gründen ausgewählt. Es gibt eine Vielzahl verschiedenartiger Heuristiken. Dadurch ergibt sich nicht nur ein unregelmäßiger Suchbaum, sondern es ist auch nicht ganz einfach, den Algorithmus mit Hilfe eines generischen Suchalgorithmus effizient zu implementieren. Seit kurzem ist eine Arbeit aus der Duke University (DOLLAS ET AL., 1995) verfügbar, die auf unveröffentlichten früheren Arbeiten aufbaut und offenbar den Stand der Technik repräsentiert (einschließlich einem ausführlicher Überblick über Anwendungen und die relevante Literatur) Mit dem dort beschriebenen Algorithmus wurde in 36000 CPU-Stunden einer Workstation (Sparc Classic) die Optimalität eines bereits bekannten Lineals mit $k = 19$ und $m = 246$ überprüft. Das im folgenden skizzierte Verfahren wurde unabhängig davon entwickelt und ist sehr ähnlich aufgebaut. Mit vergleichbarem Rechenaufwand könnten damit Probleme bis $k = 17$ oder 18 gelöst werden.

Das Grundprinzip läßt sich am leichtesten durch einen nichtdeterministischen Algorithmus beschreiben: Begonnen wird mit einem Lineal, das nur die Markierung $m_1 = 0$ trägt. Dann werden von links nach rechts Markierungen nichtdeterministisch plaziert bis alle Markierungen verbraucht sind, bis ein Abstand zwischen Markierungen mehrfach vorkommt oder bis die Länge m überschritten ist. Der Nichtdeterminismus wird durch eine Tiefensuche aufgelöst. Die folgenden Heuristiken verkleinern den Suchbaum um zwei bis drei Größenordnungen.[3)] Die Ausführungszeit wird um mindestens zwei Größenordnungen reduziert. Es bezeichne i die Anzahl noch zu plazierender Markierungen.

1. Es werden zunächst Plazierungen versucht, die möglichst kleine Abstände produzieren.

2. Mindestens $\lfloor k/2 \rfloor$ Markierungen müssen links von der Mitte liegen. Dadurch werden spiegelsymmetrische Lösungen ausgeblendet. Der Suchbaum wird dadurch mehr als halbiert, weil viele Sackgassen frühzeitig erkannt werden.

3. Die obige Symmetriebedingung läßt sich folgendermaßen verfeinern: Ist k ungerade, so darf die Markierung $m_{(k+1)/2}$ nicht rechts von der Mitte liegen. Ist k gerade, so muß Markierung $m_{k/2}$ mindestens so weit von der Mitte liegen, wie $m_{k/2+1}$.

4. Der Abstand zur nächsten neuen Markierung muß jeweils neu sein. Die Summe der i kleinsten noch nicht betrachteten Abstände ist deshalb eine untere Schranke für die restliche Länge des Lineals.

5. Zwischen den noch zu plazierenden Markierungen müssen sich $\frac{i(i-1)}{2}$ verschiedene Abstände messen lassen, die noch nicht vorgekommen sind. Der $\frac{i(i-1)}{2}$-kleinste der noch verbleibenden Abstände ist damit ebenfalls eine untere Schranke für die Restlänge des Lineals. Diese Bedingung wird nur für kleines i geprüft. Für großes i ist die folgende einfach zu testende Bedingung effektiver:

[3)] Genaue Aussagen sind schwierig, weil der Gewinn sich mit zunehmender Problemgröße erhöht, mit der Grundversion aber gar keine Probleme interessanter Größe gelöst werden können.

6. Die noch zu plazierenden Markierungen für sich betrachtet müssen selbst ein Golomblineal bilden. Aus einer Tabelle bekannter unterer Schranken für Längen kleiner Golomblineale kann bestimmt werden, ob die Einhaltung dieser Bedingung noch möglich ist.

Die Heuristiken 3–5 sind in DOLLAS ET AL. (1995) anscheinend noch nicht berücksichtigt. Für eine effiziente Implementierung einer Heuristik gilt es außerdem zu beachten, ob diese besser vor oder nach einer Knotenexpansion eingebaut wird und ob eine Beschneidung bedeutet, daß auch die folgenden Geschwisterknoten beschnitten werden können. Diese Betrachtungen sind auch von softwaretechnischem Interesse. Es zeigt sich nämlich, daß generische Suchalgorithmen, wie sie in Lehrbüchern (z.B. BRASSARD UND BRATLEY (1988)) oder zur Klassifizierung von Algorithmen (z.B.NAU ET AL. (1984)) eingesetzt werden, nicht flexibel genug sind, diese Feinheiten zu berücksichtigen. In PIGSeL wird dieses Problem dadurch gelöst, daß der Kern der Suche, nämlich Baumtraversierung, von Heuristiken und Lösungsbehandlung getrennt wird (vergleiche auch SANDERS (1994e, 1996d)).

A.2.2 0/1-Rucksackproblem

Eine Instanz des 0/1-Rucksackproblems wird durch m Gegenstände mit *Gewichten* w_i und *Profiten* p_i ($i \in \mathbb{N}_m$) sowie durch eine *Kapazität* M definiert. Es geht darum, $x_i \in \{0,1\}$ zu finden, so daß $\sum_{i<m} p_i x_i$ maximiert wird; und zwar unter Einhaltung der Bedingung $\sum_{i<m} w_i x_i \leq M$. Informell ausgedrückt soll eine Teilmenge von Gegenständen ausgewählt werden, so daß all diese Gegenstände in den Rucksack „passen“ und der durch diese Gegenstände repräsentierte Gesamtprofit maximiert wird.

Neben dem Handlungsreisendenproblem ist das Rucksackproblem wohl eins der bekanntesten und bestuntersuchten Optimierungsprobleme. Es gibt sogar ein ganzes Buch (MARTELLO UND TOTH (1990)), das sich mit dem Problem und seinen Varianten beschäftigt. Zur Vereinfachung der Sprechweise sei im folgenden immer das 0/1-Rucksackproblem gemeint, wenn vom Rucksackproblem die Rede ist. Das Problem ist **NP**-vollständig, es gibt jedoch polynomielle Approximationsalgorithmen, die beliebig gute Näherungen erlauben. Außerdem gibt es sogenannte *pseudopolynomielle* Algorithmen, die polynomiell in M und m sind. (Alle Gewichte seien in diesem Fall ganze Zahlen.) Eine ausführlich untersuchte Familie pseudopolynomieller Algorithmen beruht auf dynamischem Programmieren. Ist der Wertebereich für M und w_i jedoch groß, oder soll mit Fließkommazahlen gerechnet werden, so ist dynamisches Programmieren nur für kleine m geeignet. Ausführungszeit und vor allem Speicherplatzbedarf sind nämlich exponentiell in m.

Die besten bekannten Verfahren für den allgemeinen Fall beruhen auf Baumsuche mit Branch-and-bound-Heuristik: Zunächst werden die Gegenstände nach der Profitdichte $\frac{p_i}{w_i}$ sortiert. Im folgenden sei deshalb angenommen, daß $\frac{p_0}{w_0} \leq \cdots \leq \frac{p_{n-1}}{w_{n-1}}$. Auf Ebene i des Suchbaums wird dann entschieden, ob Gegenstand i in den Rucksack aufgenommen wird – es wird also x_i bestimmt. So entsteht ein vollständiger Binärbaum mit 2^m Blättern. Ein großer Teil des Suchbaums läßt sich jedoch durch die Branch-and-bound-Heuristik beschneiden. Eine Möglichkeit zur Berechnung einer oberen Schranke für das erzielbare $\sum p_i x_i$ beruht darauf, die Bedingung $x_i \in \{0,1\}$ zu $x_i \in [0,1]$ zu relaxieren. Das entstehende lineare Optimierungsproblem hat eine besonders einfache Struktur und läßt sich leicht in Zeit $O(m)$ lösen: Füge Gegenstände mit zunehmendem i ein, solange diese in den Rucksack passen. Das erste Element, das nicht mehr paßt, wird „zerschnitten“. Danach ist der Rucksack voll, und es gibt keine Gegenstände mit höherer Profitdichte, über die noch verfügt werden kann. Wird zu Beginn einmal ein Vorberechnungsaufwand von $O(m)$ investiert, so können sogar alle folgenden

linearen Optimierungen in Zeit $O(\log m)$ gelöst werden.[4)]

Da die Berechnung derart feinkörnig ist, ist eine Bestensuche hier nicht konkurrenzfähig, da der Aufwand zur Verwaltung einer Priority queue die Programmausführungszeit dominieren würde. Auf Parallelisierung durch Bestensuche wie in MCKEOWN ET AL. (1992) wird deshalb im folgenden auch nicht näher eingegangen. Der oben beschriebene Algorithmus entspricht dem von HOROWITZ UND SAHNI (1974) und war bis 1976 der beste bekannte Algorithmus. Von diesem wird im folgenden ausgegangen. In MARTELLO UND TOTH (1990) werden eine Reihe von Verbesserungen beschrieben, die den wesentlichen Charakter des Verfahrens aber nicht ändern.

Der Aufwand für die Suche hängt sehr von der Probleminstanz ab. In der Literatur wird i. allg. mit zufälligen Probleminstanzen gearbeitet. w_i wird dabei zufällig gleichverteilt aus einem Intervall $[w_{\min}, w_{\max}]$ gewählt. Es ist naheliegend, p_i unabhängig davon aus einem Intervall $[p_{\min}, p_{\max}]$ zu wählen. Die so erzeugten Probleminstanzen haben im Mittel nur $O(m)$ Suchbaumknoten! Die Diskussion bei der effizienten Implementierung dreht sich dann hauptsächlich darum, wie das anfängliche Sortieren der Gegenstände nach Profitdichte vermieden werden kann. Eine Parallelisierung durch Baumsuche ist dann sinnlos.[5)] Das 0/1-Rucksackproblem für unkorrelierte zufällige Werte ist ein im Mittel in linearer Zeit lösbares Problem. Um auf systematischere Weise schwierigere Instanzen erzeugen zu können, ist es deshalb üblich, p_i aus einem Intervall $[w_i + \delta_{\min}, w_i + \delta_{\max}]$ zu wählen. Durch geeignete Wahl der Parameter läßt sich die mittlere Schwierigkeit der Probleme zwischen sehr leichten Problemen und sehr schweren Problemen einstellen, bei denen exponentiell viele Knoten durchsucht werden müssen. Die sequentielle Ausführungszeit schwankt außerdem sehr stark von Probleminstanz zu Probleminstanz.

A.2.3 15-Puzzle

Das 15-Puzzle ist ein einfaches Einpersonenspiel, das in seiner mechanischen Form bis in das letzte Jahrhundert zurückreicht. In einem quadratischen Rahmen der Größe 4×4 sind 15 kleine Quadrate so angebracht, daß sich jeweils ein Quadrat verschieben läßt. Ziel ist es, mit möglichst wenigen Verschiebungen aus einer ungeordneten Anfangsstellung die in Abbildung A.3 dargestellte Zielstellung zu erreichen. In dem Beispiel wäre die kürzeste Lösung die Bewegungsfolge 9,5,4.

Das 15-Puzzle ist in der KI und der Parallelverarbeitung als Testproblem beliebt. Das Finden optimaler Lösungen für das verallgemeinerte $m \times m - 1$-Puzzle ist nach RATNER UND WARMUTH (1990) **NP**-hart. Eine offene Frage ist, wie lang der kürzeste Pfad von einer legalen[6)] Anfangsstellung zur Zielstellung im schlechtesten Fall werden kann.

Der beste bis 1992 bekannte Algorithmus zum Finden optimaler Lösungen beruht auf iterativem Vertiefen (KORF, 1985). Es findet nur die folgende einfache Heuristik Verwendung: Wenn selbst dann keine optimale Lösung gefunden werden kann, wenn alle Quadrate unabhängig voneinander auf die Zielposition bewegt werden könnten, kann die gerade betrachtete

[4)] Die veröffentlichten Laufzeiten einiger Autoren weisen darauf hin, daß diese oder ähnliche Optimierungen zwar bekannt sind, in den Veröffentlichungen aber verschwiegen werden. Deshalb sei das Verfahren hier kurz skizziert: Die partiellen Summen $\sum_{j<k} w_i$ und $\sum_{j<k} p_i$ werden in Tabellen abgelegt. Die oben beschriebene lineare Suche läßt sich dann durch eine binäre Suche ersetzen.

[5)] Die in LOOTS UND SMITH (1992) beschriebene Parallelisierung ist nur ein Scheinerfolg, da die naive Implementierung der Bewertungsfunktion mit Aufwand $\Theta(m)$ für die Knotenbewertung verwendet wurde.

[6)] Wie sich mit einem Paritätsargument leicht zeigen läßt, ist die Zielstellung nicht von jeder denkbaren Ausgangsstellung erreichbar.

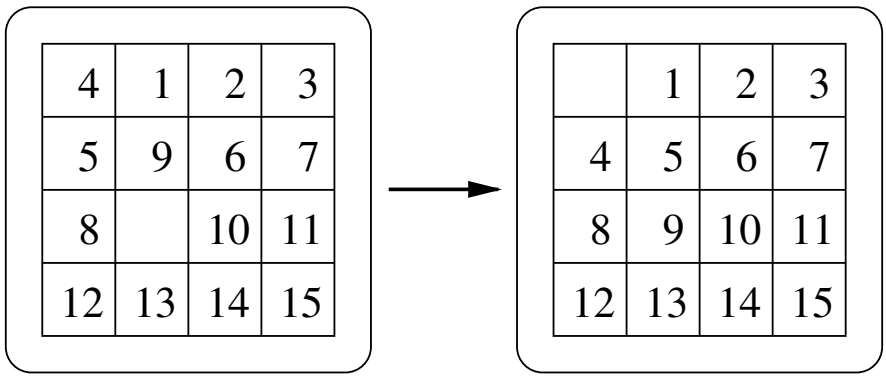

Abbildung A.3: Beispielproblem zum 15-Puzzle

Stellung verworfen werden. Die Summe der Abstände aller Quadrate von ihrem Zielquadrat gemessen in der Betragssummennorm (Manhattan-Metrik) ist also eine untere Schranke für die Anzahl noch nötiger Züge. Die Suche wird zunächst mit dieser Schranke als Maximaltiefe gestartet. Muß irgendwann ein Quadrat so bewegt werden, daß es sich von seiner Zielposition wegbewegt, so wird die Suche abgebrochen, die maximale Suchtiefe um zwei erhöht und eine neue Suche gestartet. In dieser darf einmal entgegen der Zielrichtung gezogen werden usw. Interessant an diesem Algorithmus ist, daß die Suche extrem feinkörnig ist. Auch für das $m \times m - 1$-Puzzle ist T_{atomic} eine kleine Konstante. Diese Anwendung bietet sich deshalb an, genauer zu untersuchen, welcher zusätzliche Aufwand nötig ist, um den Zustand der Suche in einem Datentyp *aufspaltbares Teilproblem* festzuhalten.

In GASSER (1995) wird eine Anzahl von Verbesserungen untersucht, die die Suche deutlich grobkörniger machen. Insbesondere werden große vorberechnete Tabellen verwendet. Der Charakter der Tiefensuche mit iterativem Vertiefen bleibt aber erhalten.

A.2.4 Firing Squad Synchronization Problem (FSSP)

In der Diplomarbeit SANDERS (1993) wurde ein Programm zur Untersuchung einer offenen Frage in der Theorie der Zellularautomaten geschrieben. Zusammengefaßt geht es darum, einen Zellularautomaten mit minimaler Zustandszahl m zu finden, der das *Firing Squad Synchronization Problem* löst. Dies geschieht durch eine Tiefensuche, bei der in jeder Ebene des Suchbaums ein Eintrag in der Übergangstabelle festgelegt wird.

Damit konnte nachgewiesen werden, daß es keine Lösung mit 4 Zuständen gibt. Das Programm wurde für den SIMD-Rechner MasPar MP-1 parallelisiert. Leider ist der Suchraum für 5 Zustände bereits so groß, daß kein Parallelrechner ihn jemals wird durchsuchen können. Sehr ähnliche Verfahren können aber verwendet werden, um andere Probleme aus der Theorie paralleler Automaten zu lösen. Zum Beispiel wurde ein *homogener Trellisautomat* mit einem Arbeitsalphabet der Größe fünf gefunden, der das Palindromproblem löst.

A.2.5 Spielbaumsuche

In der Diplomarbeit HOPP (1995) wurde untersucht, inwieweit bekannte Algorithmen für parallele Spielbaumsuche, wie sie FELDMANN (1993) beschrieben hat, skalierbar sind. Dazu wurden synthetische Spielbäume erzeugt und auf der MasPar MP-1 durchsucht. Verzwei-

gungsgrad m, Tiefe k und Aufwand für die Knotenexpansion T_{atomic} lassen sich einstellen. Bei Spielbaumsuche ist weiterhin entscheidend, mit welcher Wahrscheinlichkeit welcher Nachfolger eines Knotens den besten Zug darstellt. Dieser Parameter hat großen Einfluß auf die Effektivität der $\alpha\beta$-Heuristik. Diese wiederum führt zu Beschneidungen des Suchbaums, die vom Ergebnis der Suche in Geschwisterknoten abhängen und damit von baumförmigen Berechnungen nicht modelliert werden.

A.3 Betrachtete Maschinen

PIGSeL wurde bisher auf die folgenden Kombinationen von Maschine und Betriebssoftware portiert.

- Der Parsytec SuperCluster ist ein rekonfigurierbares Netzwerk von Transputern (T800). Als Betriebssoftware wurde das parallele Betriebssystem COSY verwendet (BUTENUTH ET AL., 1996). Für PIGSeL ist vor allem wichtig, daß COSY die asynchrone Kommunikation zwischen beliebigen PEs ermöglicht. Es wurde vor allem der lokal vorhandene SuperCluster mit 64 PEs verwendet. Einige Versuche wurden auch auf der Installation in Paderborn mit 320 PEs gemacht.[7)]

- Der Parsytec GCel-3/1024 in Paderborn [7)] ist ein 32×32 Gitter von Transputern (T805) und damit immer noch der MIMD-Rechner in Europa mit den meisten PEs. (Es gibt noch eine identische Maschhine in Köln.) Die Maschine läßt sich ebenfalls unter COSY betreiben.

- Die Maschinen Parsytec Power Xplorer und GC/PP kombinieren relativ schnelle Prozessoren vom Typ PowerPC 601 zum Rechnen, mit Transputern für die Kommunikation. Das Verbindungsnetzwerk ist ein Gitter. Die Programmierung erfolgt über das Laufzeitsystem Parix. Da Parix keinen vollständigen Satz von Funktionen für effiziente asynchrone Kommunikation besitzt, wurden diese für PIGSeL mit Hilfe der Multi-Threading Funktionalität neu implementiert.

- Für Netzwerke von Arbeitsplatzrechnern wurde eine Kommunikationsschnittstelle für PVM (GEIST ET AL., 1993) erstellt. Die Messungen wurden auf den Rechnern des Instituts vom Typ Sun SPARC durchgeführt. Diese sind durch ein Ethernet verbunden. Zeitweise standen bis zu 13 identische Maschinen für Beschleunigungsmessungen zur Verfügung.

- Eine Portierung für die Kommunikationsbibliothek MPI (SNIR ET AL., 1996) wurde bisher auf den Rechnern des Instituts unter MPICH (BRIDGES ET AL., 1995) und auf der IBM SP des Rechenzentrums mit IBMs MPI-Implementierung getestet.

Das FSSP und die Spielbaumsuche wurden auf dem SIMD-Rechner MasPar MP-1 implementiert. Die Installation am Rechenzentrum der Universität Karlsruhe besitzt 16384 PEs, die durch eine Kombination verschiedener Verbindungsnetzwerke verbunden sind. Für die Lastverteilung spielt vor allem das „Router“-Netzwerk eine Rolle, das sich als mehrstufiges Verbindungsnetzwerk klassifizieren läßt (Siehe auch MASPAR (1992); PRECHELT (1993b)).

[7)] Ich möchte dem Paderborn Center for Parallel Computing (PC^2) dafür danken, daß mir diese Maschine zur Verfügung gestellt wurde.

A.4 Details der Parallelisierung

Hier werden einige Details diskutiert, die bei der Implementierung realer Anwendungen zu beachten sind. Dabei wird von oben nach unten in der in Abschnitt A.1 skizzierten Schichtenfolge vorgegangen. Begonnen wird mit den anwendungsnahen Schichten, die in PIGSeL teilweise generisch realisiert sind. Die Abschnitte A.4.1–A.4.3 gehen auf die Realisierung der sequentiellen Suche, das Aufspalten von Teilproblemen bzw. das Packen von Teilproblemen zwecks Übertragung ein. Abschnitt A.4.4 spricht einige Optimierungsmöglichkeiten im Zusammenhang mit der Initialisierung der Suche an. Dann wird in den Abschnitten A.4.5 und A.4.6 die effiziente Implementierung von Baumbeschneidungen durch die Branch-and-bound-Heuristik und die Behandlung von Ergebnissen beschrieben. Auf die Lastverteilungsalgorithmen selbst wurde im Hauptteil der Arbeit schon ausführlich eingegangen. Die Schnittstelle zur abstrakten Maschine wird in Abschnitt A.4.7 behandelt.

A.4.1 Unterstützung der sequentiellen Suche

Sequentielle Programme zur Tiefensuche sind oft rekursiv formuliert. Der erste Schritt für die Parallelisierung besteht deshalb oft darin, das rekursive Programm in ein nichtrekursives mit expliziter Verwaltung des Stapels umzuwandeln.[8)] Leistungseinbußen sind dadurch nicht zu erwarten.[9)]

Für nichttriviale Anwendungen sind rekursive Implementierungen übrigens oft nur bedingt geeignet. Ein typisches Beispiel ist das Rucksackproblem. Wenn eine Lösung gefunden ist, so läßt sich die Menge der in den Rucksack aufzunehmenden Gegenstände aus dem Suchbaumstapel ablesen. Bei einer rekursiven Implementierung steht diese Information aber nicht zur Verfügung und muß deshalb in einer von der Anwendung verwalteten Datenstruktur ein zweitesmal aufgezeichnet werden.

Ein komplexeres Problem besteht darin, daß verschiedene Teilproblemrepräsentationen für die Operationen „sequentielle Suche“, „Aufspalten“ und „Übertragen“ unterschiedlich geeignet sind. Um eine Effizienz nahe bei eins zu erreichen, darf die innere Schleife der sequentiellen Suche nicht durch die für die Parallelisierung nötigen Datenstrukturen verlangsamt werden. Zum Beispiel kostet es beim 15-Puzzle auf dem Power Xplorer mindestens einen Faktor zwei an Geschwindigkeit, wenn wichtige Werte wie Stapelzeiger oder „aktueller Zug“ nicht in lokalen Variablen sondern in Komponenten einer Struktur gespeichert sind. Der (eigentlich gut optimierende) Motorola C-Compiler legt nämlich grundsätzlich keine Komponenten von Strukturen in Registern ab (MOTOROLA, 1993). Ähnliche Effekte sind bei vielen Compilern und Anwendungen zu erwarten. Dieses Problem läßt sich aber lösen, indem die Funktion „work“ die wichtigen Daten zu Beginn in skalare Variablen kopiert und erst bei Rückgabe der Kontrolle an den Lastverteiler die Daten in die Teilproblembeschreibung zurückschreibt. Daraus folgt für die Parallelisierung, daß die sequentielle Suche nicht einfach unterbrochen werden darf, um z.B. eine Lastaufspaltung vorzunehmen. In PIGSeL ist dies so gelöst, daß die sequentielle Suche den Lastverteiler periodisch „fragt“, ob dieser die Kontrol-

[8)] Wenn der Lastverteiler Einblick in die vom Compiler angelegten Datenstrukturen hat, kann das Programm auch rekursiv bleiben. Für High-Level Sprachen wie paralleles Prolog würde man wahrscheinlich so vorgehen. Der Lastverteiler wird dann zu einem Teil des Laufzeitsystems der Programmiersprache.

[9)] Bei SIMD-Rechnern muß man noch einen Schritt weitergehen, und die Kontrollstruktur so umformen, daß nur noch eine einzige Schleife übrigbleibt. Mit den geeigneten Methoden läßt sich dies ohne sehr großen Zusatzaufwand machen (SANDERS, 1993, 1994c,d, 1995b, 1996c; HOPP, 1995; HOPP UND SANDERS, 1995)

le übernehmen möchte. Wenn ja, wird die Teilproblembeschreibung in eine für Aufspaltung und Übertragung geeignete Form gebracht und die Funktion „work“ gibt die Kontrolle an den Lastverteiler zurück.

A.4.2 Aufspaltung von Teilproblemen

Es wurden die beiden Verfahren aus Abbildung 2.3 implementiert. Beim Golomblinealproblem wird das einfache Verfahren verwendet, bei dem jeder zweite Teilbaum auf einer Ebene möglichst nahe der Wurzel abgegeben wird. Dies funktioniert u.a. deshalb gut, weil der Suchbaum auf den obersten Ebenen einen recht hohen Verzweigungsgrad hat. Das Verfahren ist außerdem einfach und hat ein niedriges T_{split}. Ein weiterer Vorteil besteht darin, daß der Stapel aus drei klar abgegrenzten Bereichen besteht: Eine Anzahl Suchbaumebenen, in denen es keine undurchsuchten Teilbäume mehr gibt, eine jeweils einzige für die Problemaufspaltung interessante Ebene und alle darunter liegenden Ebenen werden sequentiell durchsucht. Im unteren Teil des Suchbaums kann also ein sequentieller Suchalgorithmus mit den für ihn optimalen Datenstrukturen arbeiten. Dieses einfache Verfahren funktioniert auch beim FSSP gut. Für die Spielbaumsuche wurde eine Reihe von Varianten ausprobiert, und es stellte sich als am günstigsten heraus, beim Aufspalten nur einen einzigen Nachfolger auf der obersten Ebene mit offenen Zügen abzuspalten.

Beim Rucksackproblem wäre die Aufspaltung auf einer einzigen Ebene für lösbare Probleme mit großem m ungeeignet, weil gerade auf den oberen Ebenen des Suchbaums nur sehr kleine Suchbäume abzweigen. Nur in wenigen Fällen ist es sinnvoll, ein Teilproblem mit hoher Profitdichte nicht in den Rucksack aufzunehmen. Auch das in RAO UND KUMAR (1987a) vorgeschlagene Verfahren, sehr weit unten im Stapel aufzuspalten, ist nicht gangbar, da nur wenige der Gegenstände mit niedriger Profitdichte in der optimalen Lösung auftauchen werden. Statt dessen werden Teilbäume auf jeder Ebene des Stapels abgespalten. Dazu muß der sequentielle Suchalgorithmus in der Lage sein, Teilbäume, die abgespalten wurden, als solche zu identifizieren. Weder beim Rucksackproblem noch beim 15-Puzzle ergibt sich aus dieser Anforderung eine Verlangsamung der sequentiellen Suche.

Für die Experimente zur statischen Lastverteilung wurde eine Funktion geschrieben, die sehr schnell eine Folge von Spaltoperationen auf das Wurzelproblem anwenden kann und nur ein Teilproblem zurückliefert. Es beruht auf dem in EL-DESSOUKI UND HUEN (1980) sowie SANDERS (1993) beschriebenen Verfahren. Es kommt ohne den Umweg über einen Spaltbaum aus. Außerdem spielt es keine Rolle, ob die Anzahl zu unterscheidender Teilprobleme eine Zweierpotenz ist. Das Verfahren ist um ca. eine Größenordnung schneller als die allgemeine Spaltfunktion und produziert (zumindest beim 15-Puzzle) gleichmäßigere Teilproblemgrößen.

A.4.3 Übertragung von Teilproblemen

Für die Übertragung von Teilproblemen kann eine gepackte Repräsentation nützlich sein, um Netzwerkbandbreite zu sparen. Dieses Packen kann sehr weit getrieben werden. Zum Beispiel wird in KERGOMMEAUX UND CODOGNET (1994) Abschnitt 4.3.3 eine Technik beschrieben, wie der Zustand eines kompletten Prolog-Prozesses durch einen „Orakel“ genannten Bitstring beschrieben werden kann, der nur ein Bit für jeden Stapeleintrag enthält. Dies ist deshalb möglich, weil das Wurzelproblem (in Prolog das Goal und das Programm) auf allen PEs bekannt ist. Durch redundante Berechnung kann

daraus das Teilproblem rekonstruiert werden, indem an jedem Verzweigungspunkt des Suchbaums das Orakel befragt wird, welche Verzweigung zu nehmen ist. Der Aufwand für Codierung und redundante Berechnung taucht im abstrakten Modell für baumförmige Berechnung nicht auf. Die Analysen lassen sich aber leicht anpassen. Der Aufwand wird im allgemeinen in $O(hT_{\text{atomic}})$ liegen und kann bei empfängerveranlaßten Lastverteilungsverfahren T_{split} zugeschlagen werden, da Lastübertragungen dort an Aufspaltungen gebunden sind.

A.4.4 Initialisierung

Die in dieser Arbeit beschriebenen Lastverteilungsalgorithmen beginnen damit, daß PE 0 oder alle PEs das Wurzelproblem P_{root} erhalten. In der Praxis ist zu beachten, daß etwas andere Daten zu übertragen sind als später bei der Auslieferung von Teilproblemen. Einerseits ist der Stapel bei Tiefensuche noch leer und braucht deshalb auch nicht übertragen zu werden. Andererseits gibt es Daten der Problembeschreibung, die sich nicht ändern und nur zu Beginn einmal übertragen werden müssen. Zum Beispiel müssen beim Rucksackproblem zu Beginn allen PEs die Profite und Gewichte der Gegenstände sowie die Rucksackkapazität mitgeteilt werden. Wenn diese Datenstruktur deutlich größer als eine Teilproblembeschreibung ist, muß die Analyse der Algorithmen die Kosten für einen Broadcast dieser Information zusätzlich berücksichtigen.

A.4.5 Branch-and-bound-Heuristik

Das Golomblinealproblem und das Rucksackproblem benutzen die Branch-and-bound-Heuristik. Deshalb ist es wichtig, daß die Bewertung einer neueren, besseren Lösung allen PEs so bald wie möglich bekannt gemacht wird. Leider finden in der Anfangsphase viele PEs neue Lösungen, von denen die wenigsten global optimal sind. Würde einfach bei jeder neuen Lösung ein Broadcast initiiert, könnten diese das Kommunikationsnetz verstopfen und das Gegenteil des beabsichtigten Effekts bewirken – global verbesserte Lösungen werden nur sehr langsam ausgeliefert. Deshalb sendet ein PE, das eine neue Lösung gefunden hat, dessen Bewertung zur nächsten Station auf einem Reduktionsbaum. An jedem inneren Knoten dieses Baumes wird der beste dort bekannte Wert gespeichert; neu ankommende Werte werden nur weitergeleitet, wenn sie größer als dieser Wert sind. Erst wenn an der Baumwurzel ein neuer Wert ankommt, wird dieser in einem Broadcast allen PEs mitgeteilt. Diese Technik wird bereits in KALE UND SINHA (1991) beschrieben, ist aber scheinbar relativ unbekannt.

A.4.6 Verwaltung von Lösungen

PIGSeL läßt sich für eine Vielzahl von Varianten der Behandlung von Lösungen verwenden. Es können z.B. alle Lösungen ausgegeben werden, alle besten Lösungen, nur eine beste Lösung oder jeweils die erste gefundene verbesserte Lösung. Die Suche kann auch angehalten werden sobald die erste Lösung gefunden ist. Auch Operationen wie das Zählen von Lösungen oder die Verknüpfung aller Lösungen durch eine benutzerdefinierte Operation sind möglich.

A.4.7 Schnittstelle zur Maschine

Es gibt eine Vielzahl von Details zu beachten, um eine portable Schnittstelle zu realisieren, die auf möglichst vielen Maschinen effizient läuft. Bei vielen Maschinen ist es noch nicht mal ohne weiteres möglich, eine Nachricht von einem nicht vorher spezifizierten Sender zu empfangen, ohne einen Aufwand in $\Omega(n)$ für ein Abfragen von Kommunikationsschnittstellen zu allen PEs zu investieren. In Parix muß eine effiziente Implementierung z.B. n Aktivitätsfäden auf jedem PE erzeugen – einen für jede Sender-Empfänger-Kombination.

Ein anderes Problem sind die kollektiven Operationen aus Abschnitt 2.2.3. Diese sind in neueren Bibliotheken wie MPI zwar vorhanden, aber bisher nur in einer synchronen Variante, bei der ein aufrufendes PE blockiert bis das Ergebnis berechnet ist.[10)] Für die Terminierungserkennung oder die Überwachung der globalen Last genügt dies aber zum Beispiel nicht. Deshalb enthält PIGSeL ein portables Modul für asynchrone kollektive Operationen.

A.5 Modellierung durch baumförmige Berechnungen

Anhand der implementierten Anwendungen wird nun untersucht wie adäquat das Modell baumförmiger Berechnungen ist. Dazu wird in Abschnitt A.5.1 zunächst der Frage nachgegangen, in welchen Fällen das Modell anwendbar ist, welche Abweichungen es gibt und was über die Modellparameter gesagt werden kann. Dann geht Abschnitt A.5.2 darauf ein, welche Voraussagen das Modell treffen kann, wenn es formal eigentlich nicht anwendbar ist. Wie bereits in Abschnitt 2.3.3 diskutiert, geht es dabei hauptsächlich darum, was passiert, wenn der Umfang der zu leistenden sequentiellen Arbeit von der Reihenfolge der Auswertung von Teilproblemen abhängt.

A.5.1 Modellierung der Anwendungen

Tabelle A.1 faßt die Eckdaten der betrachteten Anwendungen zusammen. Tiefe t und maximaler Verzweigungsgrad B des Suchbaums lassen sich aus den in Abschnitt A.2 eingeführten Problemparametern und der Implementierung ableiten. Beim Golomblinealproblem hängt der Verzweigungsgrad allerdings in komplizierter Weise von den Heuristiken ab, so daß nur eine obere Schranke angegeben werden kann. Die Problemgranularität T_{atomic} ergibt sich aus dem Zeitaufwand einer Knotenexpansion und ist leicht zu bestimmen. Die feinste Granularität unter den betrachteten Anwendungen hat das 15-Puzzle. Auf einem PowerPC 601 werden nur ca. 50 Taktzyklen pro Knotenexpansion benötigt. Das Golomblinealproblem mit seinen komplexen Heuristiken hat eine deutlich höhere Granularität.

Eine triviale obere Schranke für die sequentielle Ausführungszeit ergibt sich aus $T_{\text{atomic}} \cdot B^t$. Durch die verwendeten Heuristiken kann die reale Ausführungszeit aber deutlich geringer ausfallen. Vor allem beim Rucksackproblem kann die Anzahl expandierter Knoten je nach Probleminstanz zwischen linear und exponentiell schwanken. Bei der Spielbaumsuche wird der mittlere Verzweigungsgrad durch die $\alpha\beta$-Heuristik beinahe halbiert. (Zumindest bei guter Zugsortierung.)

[10)] Für MPI-2 sind offenbar asynchrone kollektive Operationen geplant (MPI FORUM, 1996).

Tabelle A.1: Überblick über Problem- und Modellparameter für die Beispielanwendungen. „1. Lsg.“=Suche stoppt sobald erste Lösung gefunden; „Beschn.“= Sind Heuristiken zur Beschneidung des Suchbaums von der Auswertungsreihenfolge abhängig? B&B=Branch-and-bound-Heuristik. Einklammerung bedeutet, daß der Effekt nicht bei allen Probleminstanzen auftritt.

Eigenschaft	Golomb-lineale	Rucksack-problem	m^2-1-Puzzle	FSSP	Spielbaum-suche
Tiefe t	k	m	$\mathrm{O}(m^3)$	$\approx m^3$	k
Breite B	$\mathrm{O}(m)$	2	3	$m-1$	m
T_{atomic}	$\Theta(m)$	$\mathrm{O}(\log m)$	$\mathrm{O}(1)$	$\mathrm{O}(1) \gg 1$	T_{atomic}
T_{seq}	$mm^{\mathrm{O}(k)}$	$\mathrm{O}(m\log m)$– $\mathrm{O}(2^m \log m)$	$2^{\mathrm{O}(m^3)}$	$\mathrm{O}\left(m^{m^3}\right)$	$\Omega(T_{\text{atomic}} \cdot (m/2)^k)$
T_{split}	$\Theta(m)$	$\mathrm{O}(m)$	$\mathrm{O}(m^3)$	$\mathrm{O}(m^3)$	$\mathrm{O}(k\log m)$
h	$\leq k\lceil\log m\rceil$	$\leq m$	$\mathrm{O}(m^3)$	$\mathrm{O}(m^3)$	$\Omega(km)$
l	$\Theta(m)$	$\mathrm{O}(m)$	$\mathrm{O}(m^3)$	$\mathrm{O}(m^3)$	$\mathrm{O}(k\log m)$
1. Lsg.	(Ja)	(Ja)	(Ja)	Nein	Nein
Beschn.	(B&B)	(B&B)	Nein	Ja	Ja

Der Problemaufspaltungsaufwand T_{split} und die Nachrichtenlänge l sind in allen betrachteten Anwendungen proportional zur Größe der Datenstruktur für die Suche. Die größte Rolle spielt dabei meist der Stapel.

Die maximale Spalttiefe h hängt von Tiefe und Verzweigungsgrad des Suchbaums einerseits und der verwendeten Aufspaltfunktion andererseits ab. Wird nach der einfacheren der in Abschnitt A.4.2 beschriebenen Methoden gespalten, wie beim Golomblineal und dem FSSP, so läßt sich h durch $t\lceil\log B\rceil$ nach oben abschätzen. Wird auf allen Ebenen gleichzeitig gespalten, ist h möglicherweise kleiner. Das ist vor allem beim Rucksackproblem wichtig, wo der Suchbaum sehr tief werden kann. Bei der Spielbaumsuche hat es sich als günstig herausgestellt, nur einen einzigen Teilbaum möglichst weit oben im Suchbaum abzuspalten. Außerdem werden nicht alle Teilbäume sofort zur Aufspaltung freigegeben, so daß h recht groß werden kann.

Bei allen betrachteten Anwendungen, außer leicht zu lösenden Instanzen des Rucksackproblems, wachsen die Parameter h, T_{atomic}, l und T_{split} viel langsamer als T_{seq}. Für hinreichend große Probleme ist deshalb mit einer effizienten parallelen Ausführung zu rechnen, falls das Modell baumförmiger Berechnungen zutrifft. Dies ist beim Golomblinealproblem, dem Rucksackproblem und dem 15-Puzzle gegeben, wenn das Programm mit einer Schranke für die Lösungsqualität aufgerufen wird, die keine Lösung zuläßt. Es handelt sich dann um Widerlegungssuchen im Sinne von Abschnitt 2.3.3. Bei iterativem Vertiefen, wie es für das 15-Puzzle und eine mögliche Einsatzweise der Golomblinealsuche verwendet wird, sind alle Iterationen bis auf die letzte Widerlegungssuchen. Bei der letzten Iteration wird die Suche gestoppt sobald eine Lösung gefunden ist. Wird beim Golomblineal oder dem Rucksackproblem ohne eine genaue Schranke für die Lösung gestartet, so kann die Entdeckung einer Lösung zur Beschneidung von Teilbäumen führen. Nachdem die beste Lösung gefunden ist, verhält sich der Rest der Suche wie eine Widerlegungssuche.

Während die Entdeckung einer Lösung bei Branch-and-bound oder iterativem Vertiefen sich überall im Suchbaum auswirkt, gibt es beim FSSP und bei der Spielbaumsuche lokal wirkende Heuristiken. Wenn die Suche beim FSSP in eine Sackgasse läuft, wird überprüft, ob die Rücknahme der letzten Entscheidung ausreicht, um die Sackgasse zu verlassen. Wenn nicht, findet ein weiterer Backtracking-Schritt statt usw. Dies kann dazu führen, daß die Auswertung eines Teilbaums die Auswertung eines anderen überflüssig macht. Wurde dieser überflüssige Teilbaum abgespalten, wird er trotzdem durchsucht.

Bei der Spielbaumsuche führt die $\alpha\beta$-Heuristik dazu, daß Teilbäume nicht durchsucht werden, wenn bereits feststeht, daß ihre Auswertung den spieltheoretischen Wert der Stellung nicht mehr ändern kann. Ob dies der Fall ist, hängt von der Auswertung der Geschwisterteilbäume ab.

Insgesamt läßt sich sagen, daß baumförmige Berechnungen sich gut zu realen Anwendungen in Beziehung setzen lassen. Über die Modellparameter lassen sich oft gute Abschätzungen durch Inspektion des Anwendungscodes herleiten. Die verbleibenden Parameter können durch Messungen repräsentativer Probleminstanzen abgeschätzt werden. Dazu ist kein Parallelrechner erforderlich. Die Annahme der Unabhängigkeit von Teilproblemen ist für viele, aber leider nicht alle Anwendungen ganz oder teilweise gerechtfertigt.

A.5.2 Was geschieht bei abhängigen Teilproblemen?

Die Tatsache, daß baumförmige Berechnungen eine Anwendung nicht vollständig modellieren, bedeutet noch nicht, daß das Modell völlig nutzlos ist. Im besten Fall kann es sich herausstellen, daß die Abhängigkeiten unerheblich sind. Zum Beispiel zeigt es sich, daß der parallele Algorithmus für das FSSP kaum mehr Knoten als der sequentielle Algorithmus durchsucht. Die in Abschnitt A.5.1 beschriebene Heuristik greift nämlich vor allem in den unteren Ebenen des Suchbaums, während die Aufspaltung von Teilproblemen hauptsächlich in den oberen Ebenen stattfindet.

Tabelle A.2 zeigt sequentielle Ausführungszeiten für Golomblinealprobleme mit 10–13 Markierungen. Für m wurde jeweils die optimale Länge, die optimale Länge minus eins und eine große Länge gewählt, die oberhalb analytischer oberer Schranken liegt. Gemessen wurde die Anzahl betrachteter Suchbaumknoten und die CPU-Zeit auf einer Sun SS/4 mit 110MHz mit Gnu-C 2.7. Wird das optimale m bereits als Schranke eingegeben, so wird ziemlich schnell eine Lösung gefunden – der Löwenanteil des Zeitaufwandes wird dann dafür benötigt, die Optimalität dieser Lösung zu beweisen. Deshalb ist die Ausführungszeit für die erfolglose Suche mit kleinerem m auch kaum kleiner. Wird dagegen keine gute Schranke für m angegeben, ist eine deutliche, aber begrenzte Vergrößerung des Suchbaums beobachtbar. Zumindest bei Benutzung von iterativem Vertiefen und nicht zu großer Anzahl PEs läßt sich schon aus diesen Daten auf eine effiziente Parallelisierbarkeit schließen. Auch mit vielen PEs ergibt sich eine recht gute Beschleunigung. Für 12 bzw. 13 Markierungen und $m = 100$ bzw. $m = 150$ wurden auf 256 PEs des GCel durchschnittliche Beschleunigungen von 164 bzw. 184 gemessen. Das ist zwar schlechter als die Beschleunigungen von 217 bzw. 252, die sich für die entsprechenden Widerlegungssuchen ergeben (siehe auch Abbildung A.5), aber immer noch recht gut.

Beim Rucksackproblem sind die Auswirkungen der Branch-and-bound-Heuristik auf die parallele Suche stärker. Abbildung A.4 zeigt die auf 1024 PEs des GCel erzielte

Tabelle A.2: Sequentielle Ausführungszeit und Suchbaumgröße für das Golomblinealproblem (k: Anzahl Markierungen, m: Schranke für Lineallänge).

k	m	T_{seq}	#Knoten
10	54	0.22	35673
10	55	0.24	36617
10	70	0.31	46337
11	71	5.19	638408
11	72	5.14	641999
11	100	5.46	671410
12	84	13.66	1685701
12	85	14.85	1807307
12	100	30.44	3333915
13	105	330.00	31394784
13	106	370.30	34914682
13	150	568.27	51448898

Beschleunigung für 256 Instanzen mit $m = 2000$, zufälligen $w_i \in [0.01, 1.01]$ und $p_i \in [w_i + 0.1, w_i + 0.125]$ sowie $M = \sum w_i/2$. Diese Parameter wurden gewählt, um Probleminstanzen mit großem m zu erhalten, die sequentiell lösbar sind, aber genügend schwierig sind, um eine Parallelisierung sinnvoll erscheinen zu lassen. Wegen der großen Nachrichtenlänge $l \in \Theta(m)$ und der geringen Breite des Suchbaums, ergeben sich Probleminstanzen, die eine gewisse Herausforderung an die Parallelisierung stellen. Sowohl die sequentiellen Ausführungszeiten als auch die erreichten Beschleunigungen schwanken um mehrere Größenordnungen. Es gibt viele sehr kleine Probleme, für die keine hohe Effizienz erwartet werden kann. Ab einer sequentiellen Ausführungszeit von $10 \cdot 1024s$ sind hohe Effizienzen beobachtbar. Für sehr große Probleme wird eine Effizienz $\gg 1$ beobachtet. Für diese Instanzen läuft die sequentielle Suche offenbar in eine Sackgasse, aus der sie lange nicht herausfindet. Die parallele Suche ist robuster, da sie mehrere Pfade gleichzeitig ausprobiert. Werden alle sequentiellen und parallelen Ausführungszeiten summiert, so stellt sich heraus, daß das parallele Programm 1410 mal schneller ist als das sequentielle. Offenbar ist über die sequentielle Suchstrategie für das Rucksackproblem noch nicht das letzte Wort gesprochen. Es macht Sinn, auch dort mehrere Bereiche des Suchbaums (quasi)-gleichzeitig im Auge zu behalten.

Bei allen bisher betrachteten Anwendungen sind die Auswirkungen von Abhängigkeiten zwischen Teilbäumen auf die Effizienz der Suche also relativ gering, z.T. sogar positiv. Ganz anders sieht es für die Spielbaumsuche aus. Es läßt sich leicht zeigen, daß bei naiver Anwendung der Lastverteilungsalgorithmen für baumförmige Berechnungen die Beschleunigung nur in $O(\sqrt{n})$ liegt. Durch möglichst frühzeitiges Beenden nutzlos gewordener Teilbaumauswertungen und durch Heuristiken, die steuern, welche Teilbäume wann ausgewertet werden, läßt sich die Situation aber verbessern. Das Modell baumförmiger Berechnungen kann bei der Analyse dieser Heuristiken nützliche Dienste leisten. Diese bewirken nämlich eine Vergrößerung der maximalen Spalttiefe h. Dadurch wird der Suchbaum unregelmäßiger und die Lastverteilung schwieriger.

Auch bei anderen Anwendungen können baumförmige Berechnungen herangezogen wer-

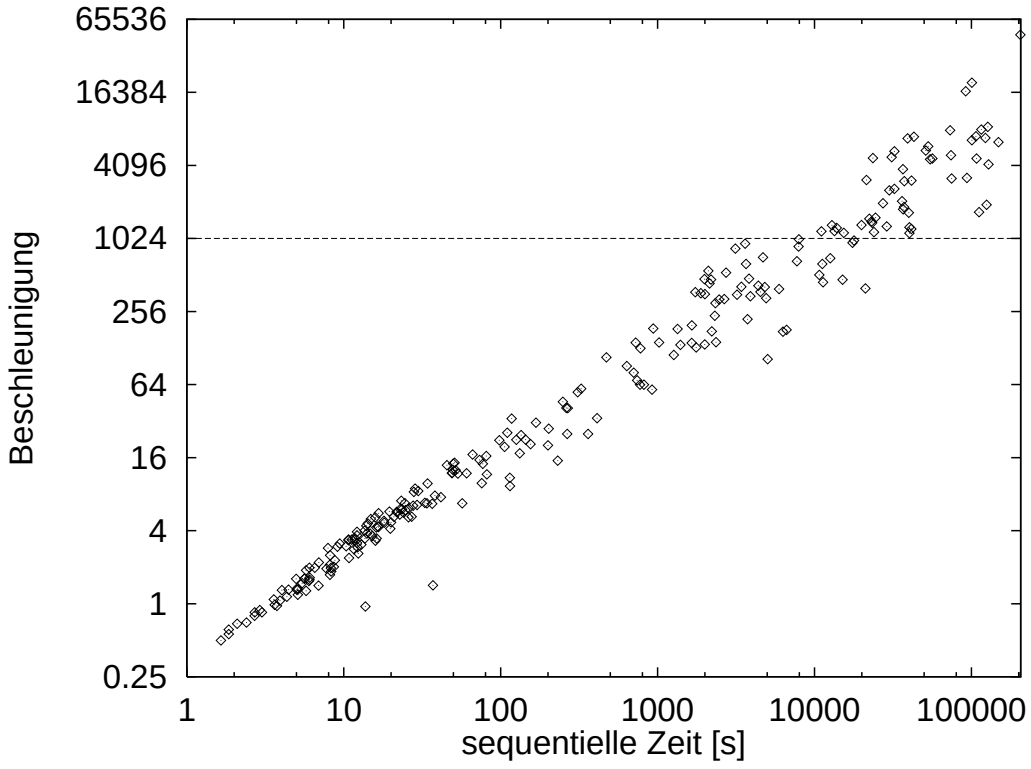

Abbildung A.4: Beschleunigung für 256 zufällige Instanzen des Rucksackproblems auf 1024 PEs.

den, um die Lastverteilungsaspekte der Anwendung zu modellieren. Zusätzlich muß allerdings geklärt werden, wie sich redundante Berechnungen oder zusätzliche Kommunikationsoperationen auf die Gesamtleistung auswirken.

A.6 Wie gut ist zufälliges Anfragen?

Wie bereits erwähnt hat sich zufälliges Anfragen schon für eine Vielzahl von Anwendungen bewährt. Ergebnisse für viele Prozessoren werden in ARVINDAM ET AL. (1990a) angegeben. Auf einem Hyperwürfel mit 1024 Prozessoren (Ncube/10) werden Messungen mit „Satisfiability“-Problem gemacht. Beschleunigungen zwischen 564 bei einer parallelen Ausführungszeit von 113s und 1007 bei einer parallelen Ausführungszeit von 6665s werden für verschieden große unerfüllbare Formeln gemessen. Offen bleibt dabei die Frage, ob auch für kleine Probleminstanzen gute Beschleunigungen erreicht werden können und ob zufälliges Anfragen auch auf Verbindungsnetzwerken mit großem Durchmesser trotz der globalen Kommunikation gut funktioniert.

Um dieser Frage nachzugehen, wurden Messungen für das Golomblinealproblem auf dem GCel durchgeführt. Auf diesem Rechner wurden mit anderen Lastverteilungsalgorithmen schon vorher hohe Beschleunigungen erreicht, allerdings nur für relativ große Probleme. In REINEFELD UND SCHNECKE (1994) werden für schwere Instanzen des 15-Puzzles Beschleunigungen um 900 bei parallelen Ausführungszeiten im Minutenbereich erreicht.[11)] Für das

[11)] Dabei wurde über diejenigen 25 von 100 Probleminstanzen mit der größten sequentiellen Ausführungszeit ge-

$4 \times 5 - 1$-Puzzle auch noch höhere Beschleunigungen bei Ausführungszeiten von ca. 30min. In TSCHÖKE ET AL. (1994) wird Bestensuche für das Handlungsreisendenproblem betrachtet. Hier wird bei eine Effizienz um 75 % bei parallelen Ausführungszeiten von ca. 10min erreicht.

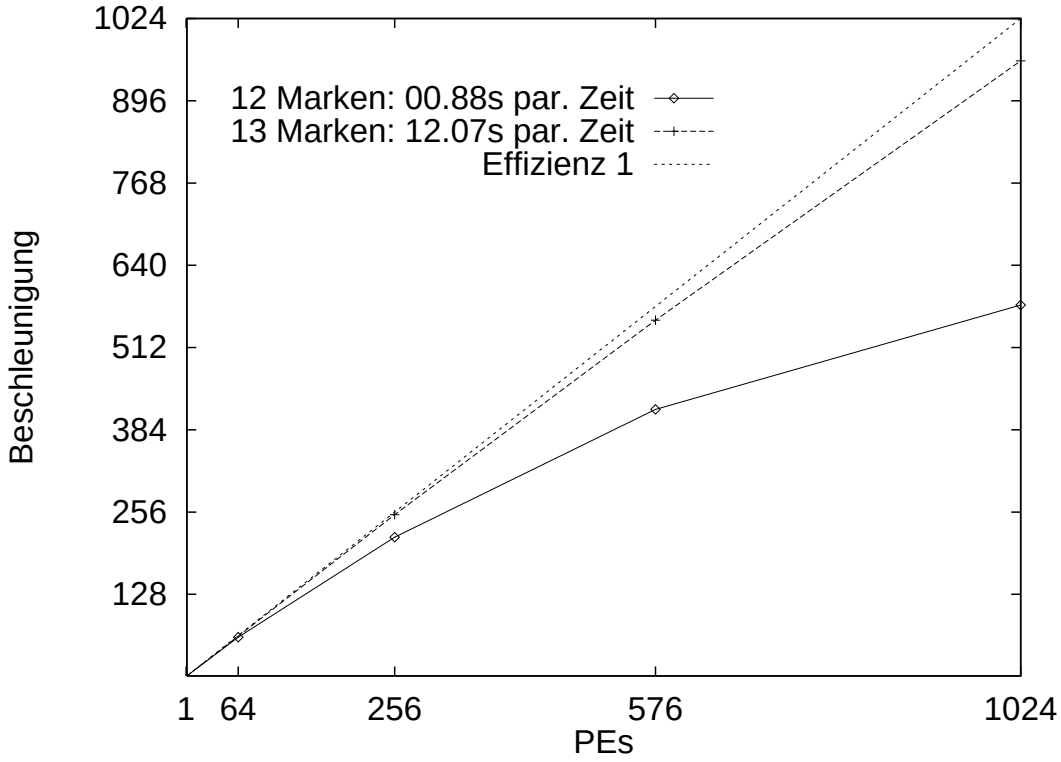

Abbildung A.5: Beschleunigung für Golomblinealprobleme auf GCel.

Abbildung A.5 zeigt Beschleunigungsmessungen für Golomblineale mit 12 und 13 Markierungen. Die Maximallänge wurde so gewählt daß gerade keine Lösung gefunden wird – es wird also die Optimalität der optimalen Lösung überprüft. Da es sich um reine Widerlegungssuchen handelt, ist das Modell baumförmiger Berechnungen hier anwendbar. Die Zeit für das Programmladen wird nicht mitgerechnet. Die sequentiellen Ausführungszeiten sind mit einer Variante des parallelen Programms gemessen, das für den Fall $n = 1$ vernachlässigbaren Kommunikationsaufwand hat. Um die (geringen) Laufzeitschwankungen durch den randomisierten Lastverteiler auszugleichen, wurde über 5 Messungen gemittelt. Selbst das kleine Problem mit 12 Markierungen erreicht eine Beschleunigung von 578 bei einer parallelen Ausführungszeit von $0.88s$. Das Lineal mit 13 Markierungen wird in $12.07s$ überprüft, 958 mal schneller als durch das sequentielle Programm.

Zufälliges Anfragen funktioniert genauso gut am anderen Ende des Spektrums von Parallelrechnern. Abbildung A.6 zeigt die Beschleunigung für die gleichen Probleminstanzen wie oben für 13 durch ein Ethernet verbundene Arbeitsplatzrechner vom Typ Sun SPARC SLC. Trotz geringerer Prozessorzahl tritt eine hohe Effizienz erst bei größerer Last pro PE

mittelt. Daraus ergibt sich leider ein methodisches Problem. Es könnte sein, daß diese Instanzen nicht nur wegen ihrer Größe leicht zu parallelisieren sind, sondern auch weil die sequentielle Bearbeitung in eine Sackgasse gelaufen ist. (Siehe auch die Diskussion in Abschnitt A.5.2.) Hier wird dieses Problem vermieden, indem nur erfolglose Iterationen der Suche betrachtet werden, die keine solchen Beschleunigungsanomalien aufweisen können.

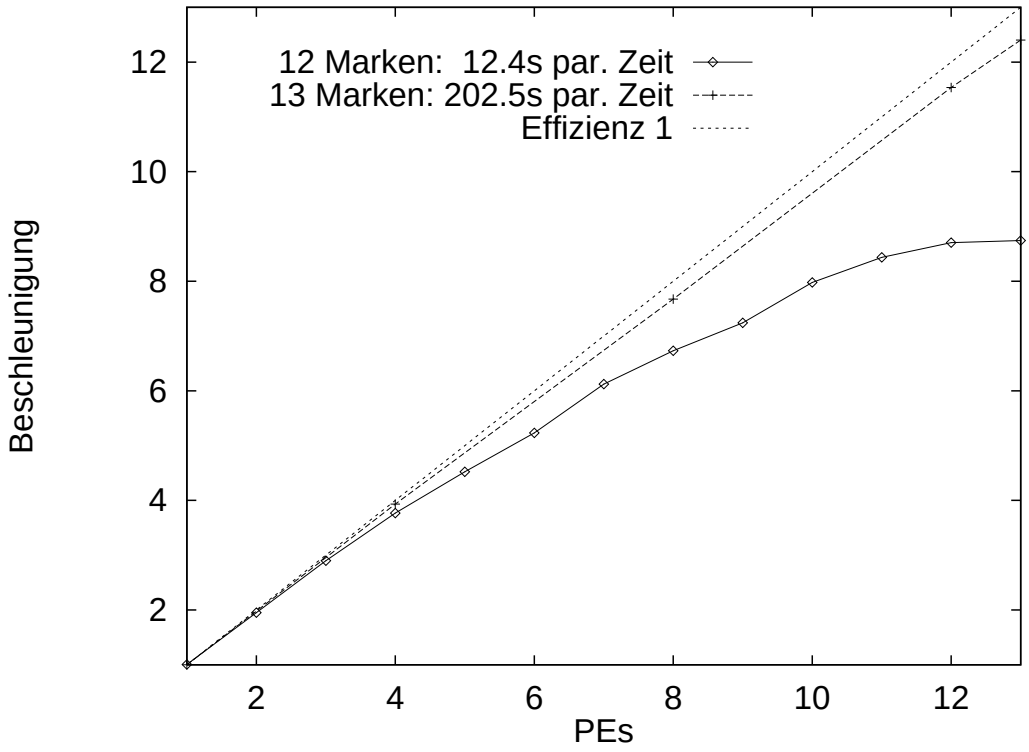

Abbildung A.6: Beschleunigung für Golomblinealprobleme auf lokalem Netzwerk.

auf als bei der transputerbasierten Maschine, bei der Kommunikations- und Rechenleistung besser zusammenpassen. Trotzdem zeigt sich, daß lokale Netzwerke für die Parallelisierung baumförmiger Berechnungen durchaus geeignet sind.

Empfängerveranlaßte Lastverteilungsalgorithmen haben auf Netzen von Arbeitsplatzrechnern noch einen weiteren Vorteil: Alle PEs werden gleichmäßig ausgelastet, selbst wenn die beteiligten Rechner unterschiedlich schnell sind. Dies gilt sogar, wenn die für die Suche zur Verfügung stehende Rechenleistung durch den Einfluß anderer Benutzer dynamisch variiert. Solange das Netzwerk nicht völlig blockiert ist und die Suchprozeße hinreichend oft die CPU bekommen, um noch Lastanfragen beantworten zu können, wird sich für hinreichend große Probleme eine gute Lastverteilung einstellen. Dies wurde für große Instanzen des Golomblinealproblems auch auf den Arbeitsplatzrechnern der Lehrstuhls ausprobiert und funktionierte reibungslos. (Genauere, quantitative Aussagen sind allerdings schwierig.)

A.6.1 Wie gut ist die SIMD-Variante?

Für das FSSP wird ein Vorläufer des hier betrachteten synchronen Algorithmus für zufälliges Anfragen erfolgreich verwendet. Allerdings sind die in Abschnitt 2.4.2 beschriebenen auf Präfixsummenbildung basierenden Verfahren noch etwas effizienter. Der Hauptgrund dafür ist, daß Teilproblembeschreibungen so lang sind, daß die Übertragung der Teilprobleme deutlich länger dauert, als die komplexen Berechnungen, die angestellt werden, um allen PEs mit einer einzigen Lastübertragungsphase Arbeit zu verschaffen.

Bei den Spielbaumsuchalgorithmen ist dagegen zufälliges Anfragen im allgemeinen überlegen. Auf Grund der höheren Irregularität des Problems kann ohnehin keine hohe PE-Auslastung erreicht werden. Außerdem sind Teilproblembeschreibungen kürzer. Hier kommt

es deshalb darauf an, sich möglichst oft eine Lastverteilung leisten zu können, um die vielen kleinen Teilbäume schnell durch neue Arbeit ersetzen zu können.

A.7 Wieviel bringen statische Lastverteilung und Initialisierung?

In Kapitel 8 wurde auf Grund analytischer Überlegungen vorausgesagt, daß statische Lastverteilung am ehesten mit dynamischen Verfahren konkurrieren kann, wenn besonders kleine Probleminstanzen zu lösen sind, bei denen kein Verfahren eine sehr hohe Effizienz erreichen kann. Um dieser Vermutung nachzugehen, wurden sämtliche erfolglosen Iterationen des 15-Puzzle Algorithmus mit den in KORF (1985) veröffentlichten Testproblemen betrachtet. Weil 100 unterschiedlich schwierige Anfangsstellungen für mehrere Iterationstiefen zu untersuchen sind, ergeben sich 800 Probleminstanzen, deren sequentielle Ausführungszeiten sich über 8 Größenordnungen erstrecken. (Auf einem 77MHz Power2 Prozessor zwischen 17 μs und 2389 s.)

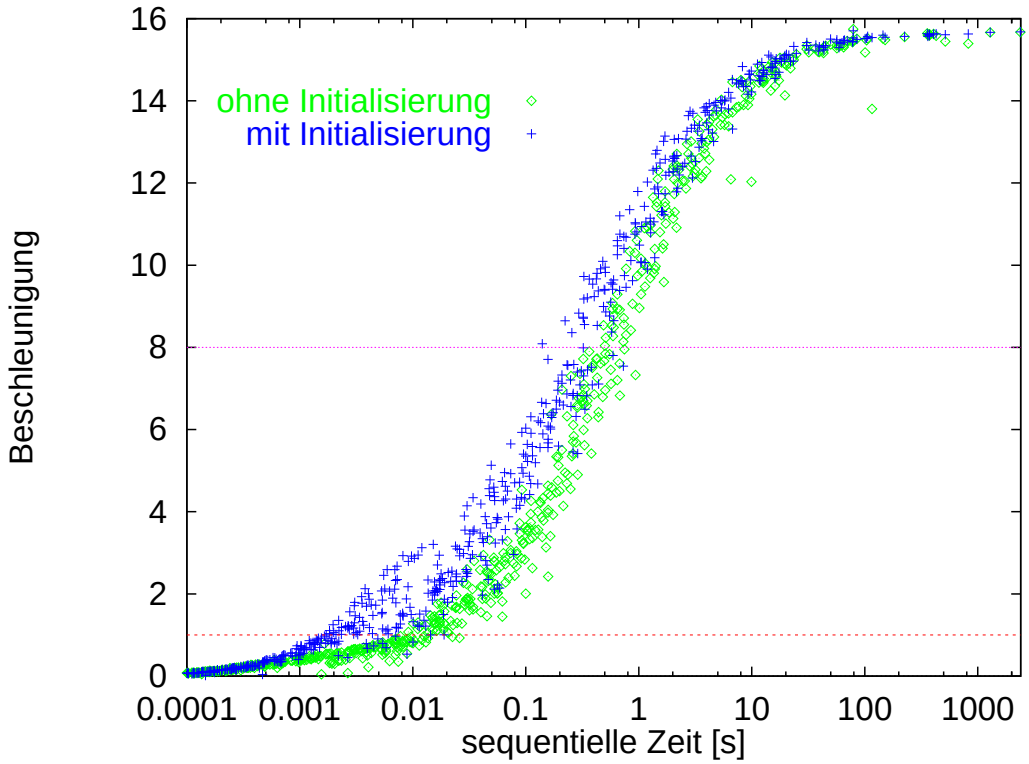

Abbildung A.7: 15-Puzzle Instanzen und zufälliges Anfragen. (16 PEs.)

Abbildung A.7 zeigt sequentielle Ausführungszeit und Beschleunigung dieser Instanzen auf 16 Wide-Nodes der IBM-SP der Universität Karlsruhe[12)] für zufälliges Anfragen mit und ohne Initialisierung. Die Initialisierung bringt in jedem Fall einen Vorteil, der bei kleinen Probleminstanzen mit $T_{\text{seq}} \leq 0.1$ s sehr deutlich ausfällt, allerdings starken Schwankungen unterliegt. Mit Initialisierung ist schon bei sequentiellen Ausführungszeiten zwischen 2 und 3

[12)] Weitere Informationen finden sich unter `http://www.uni-karlsruhe.de/~SP`

ms eine Beschleunigung größer eins üblich, während dies ohne Initialisierung erst bei 10 bis 20 ms sequentieller Ausführungszeit der Fall ist. Bei iterativem Vertiefen kann deshalb schon früher von sequentieller zu paralleler Bearbeitung übergegangen werden.

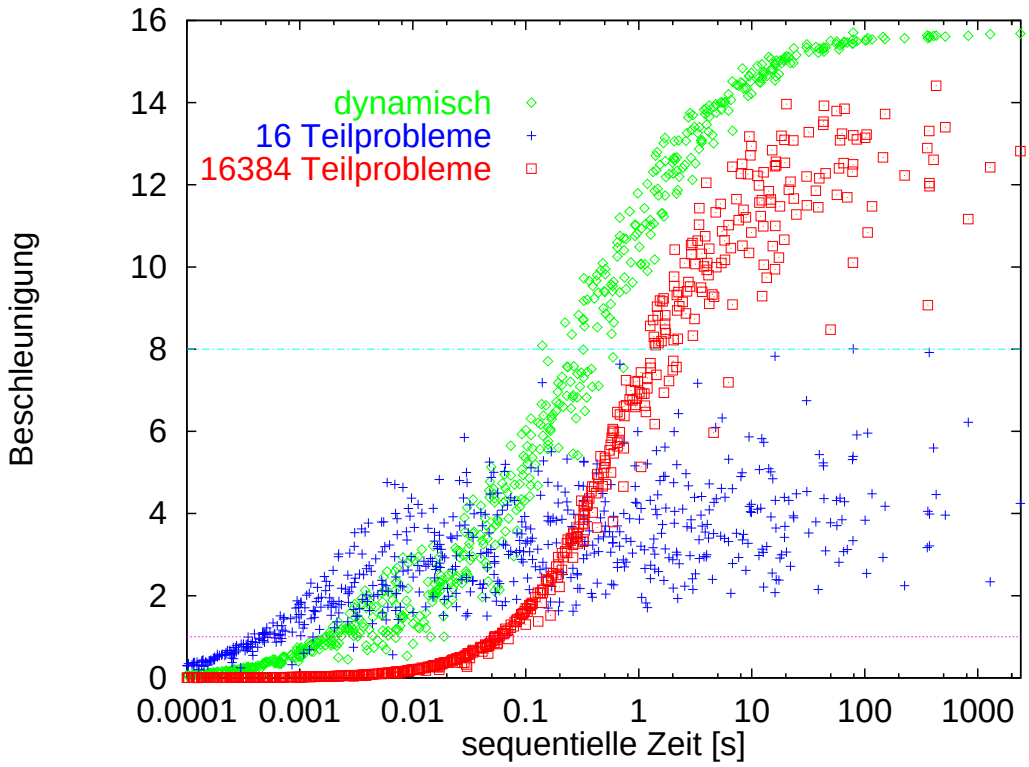

Abbildung A.8: Statische versus dynamische Lastverteilung für das 15 Puzzle. (16 PEs.)

Noch kleinere Instanzen können sinnvoll parallel bearbeitet werden, wenn reine statische Lastverteilung eingesetzt wird, da dann das Protokoll für Terminierungserkennung und Lösungsbearbeitung drastisch vereinfacht werden kann. Abbildung A.8 vergleicht die Leistung von zufälligem Anfragen mit Initialisierung mit der von statischer Lastverteilung bei einem bzw. 1024 Teilproblemen pro PE. Für sequentielle Ausführungszeiten kleiner 10 ms ist die reine statische Lastverteilung deutlich schneller als das dynamische Verfahren. Schon bei Ausführungzeiten von ca. 400μs ist eine parallele Bearbeitung sinnvoll. Effizienzen nahe eins sind mit statischer Lastverteilung aber nur zu erreichen, wenn sehr viele Teilproblem erzeugt werden. Der Aufwand dafür ist auf Maschinen mit schneller Kommunikation – wie der IBM SP – viel größer als der zusätzliche Kommunikationsaufwand dynamischer Verfahren. Außerdem bleibt die Beschleunigung sehr stark von der Probleminstanz abhängig, während randomisierte dynamische Lastverteiler bei großen Probleminstanzen sehr zuverlässig sind.

Abbildung A.9 zeigt, daß es trotzdem sinnvoll sein kann, mehr als ein Teilproblem pro PE zu erzeugen. Im Bereich zwischen 10 und 100 ms ist z.B. die Erzeugung von 16 Teilproblemen pro PE schneller als die dynamische Lastverteilung und schneller und zuverlässiger als die Erzeugung von nur einem Teilproblem pro PE.

Soll das „letzte“ aus einem Algorithmus für paralleles iteratives Vertiefen herausgeholt werden, bietet es sich also an, das eingesetzte Verfahren (sequentiell, statisch, dynamisch) und dessen Parameter (Anzahl erzeugter Teilprobleme) von einer heuristischen Voraussage

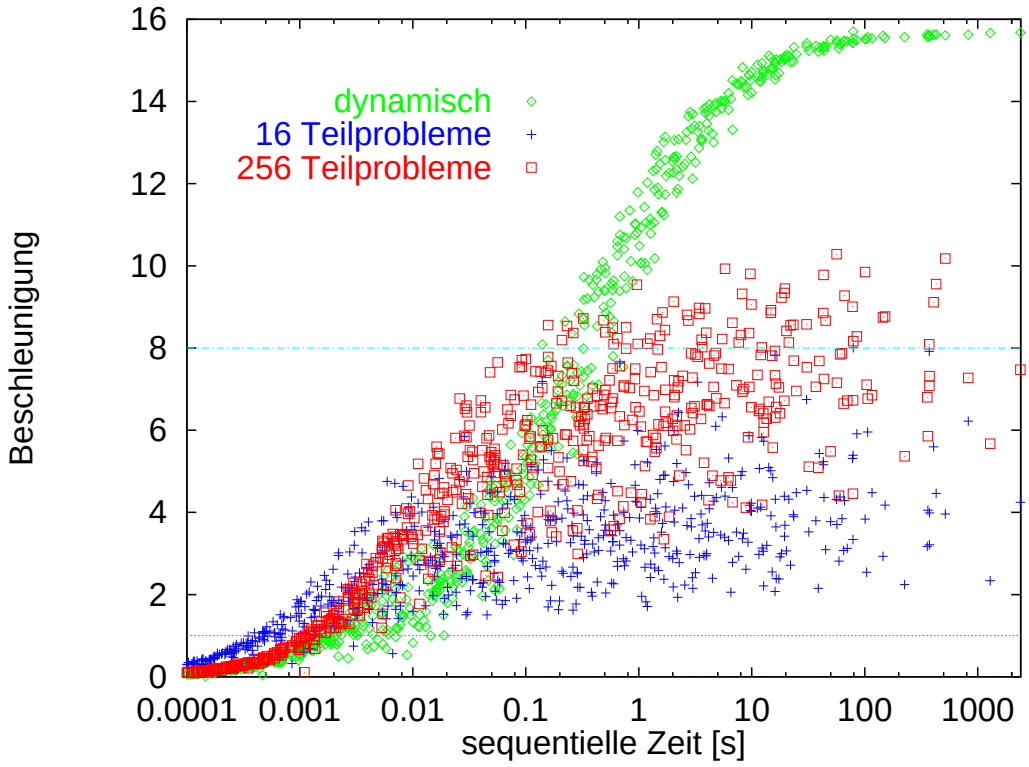

Abbildung A.9: Nochmal statische versus dynamische Lastverteilung. (16 PEs.)

über die Problemgröße abhängig zu machen. Beim 15-Puzzle ließe sich eine solche Funktion z.B. aus der Manhattan-Distanz und der Iterationsnummer von iterativem Vertiefen ableiten.

A.8 Einsparung globaler Kommunikationen

Zufälliges Anfragen ist für einigermaßen große Probleminstanzen kaum zu schlagen, da es dort eine Effizienz nahe eins erreicht. Für sehr kleine Instanzen, bei denen nur eine geringe Effizienz erreichbar ist, bieten sich statische Verfahren als Ergänzung an. Bleibt die Frage, ob im Bereich dazwischen reale Verbesserungen durch Techniken wie Fragen-und-Mischen oder lokales zufälliges Anfragen erreichbar sind. Zu diesem Zweck wurde folgende Variante von zufälligem Anfragen implementiert:

- Es wird das Initialisierungverfahren aus Abschnitt 8.4.1 verwendet. Dabei wird eine Pseudozufallspermutation auf der Basis von primitiven Polynomen verwendet (siehe auch Abschnitt 4.3.2).

- Es werden keine weiteren Zufallspermutationen durchgeführt.

- Jedes PE verwaltet einen Phasenzähler i, der nicht mit den anderen PEs synchronisiert ist, sondern bei jeder Lastanfrage erhöht wird. Eine Lastanfrage wird innerhalb eines Kommunikationsradius abgewickelt, der so gewählt wird, daß ca. $2^{i \bmod \log n}$ PEs erreichbar sind. Dieses Anfrageschema sei *exponentielles zufälliges Anfragen* genannt.

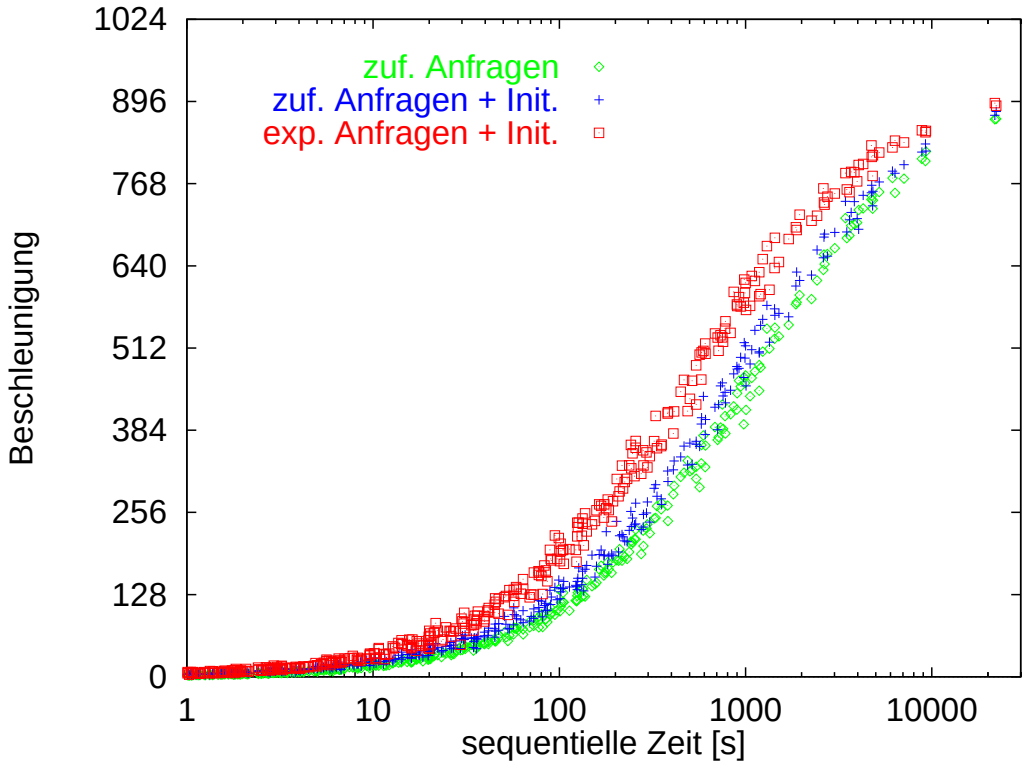

Abbildung A.10: Beschleunigungen für verschieden große 15-Puzzle-Instanzen ohne Initialisierung, mit Initialisierung und mit exponentiellem Anfragen. (1024 PEs.)

Für die Messung auf dem Parsytec GCel mit 32×32 PEs wurden die gleichen Instanzen des 15-Puzzle verwendet wie in Abschnitt A.7. Auf die allergrößten Instanzen wurde allerdings verzichtet, da die Messung der sequentiellen Ausführungszeiten auf einem Transputer sehr lange gedauert hätte. Abbildung A.10 zeigt die erreichte Beschleunigung für globales zufälliges Anfragen ohne Initialisierung, mit Initialisierung und für exponentielles zufälliges Anfragen mit Initialisierung. Exponentielles zufälliges Anfragen bewirkt eine erkennbar verbesserte Beschleunigung, die vor allem im Bereich mittlerer Problemgrößen zu einer deutlich höheren Effizienz führt.

Selbst bei sehr großen Probleminstanzen bleibt die Beschleunigung unter 900. Das liegt daran, daß die Abfragehäufigkeit Δt relativ klein eingestellt wurde, um eine möglichst hohe Effizienz bei mittleren Problemgrößen zu erreichen.[13)] Für ein zehnmal so großes Δt ergibt sich eine Beschleunigung bis zu 960.

Abbildung A.11 zeigt das Verhältnis der parallelen Ausführungszeiten von globalem und exponentiellem zufälligen Anfragen (beide mit Initialisierung). Bei Problemen mit sequentiellen Ausführungszeiten um die 100 s ergibt sich eine Verbesserung von 40–50 %. Auch bei kleineren Problemen gibt es eine ähnliche Verbesserung, diese schwankt aber stärker und ist außerdem nicht so relevant, da trotz allem keine hohe Effizienz erreicht wird.

[13)] Eine automatische Anpassung von Δt an die (geschätzte) Problemgröße wäre übrigens (zumindest für das 15-Puzzle) kein großes Problem. Trotzdem ist es ärgerlich, daß erfolgloses versuchendes Empfangen in COSY – wie auch bei anderen Systemen – unnötig ineffizient implementiert ist.

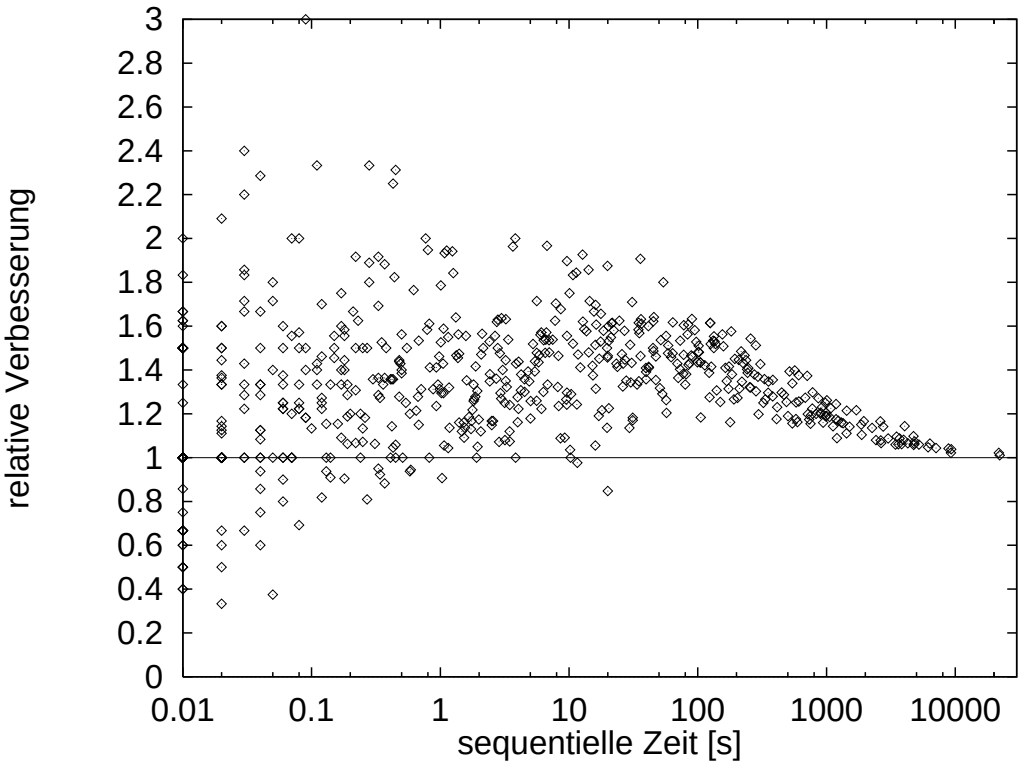

Abbildung A.11: Relative Verbesserung von exponentiellem zufälligen Anfragen gegenüber globalem zufälligen Anfragen (beides mit Initialisierung).

A.9 Zusammenfassung

Es konnte anhand einiger einfacher aber nichttrivialer Anwendungen gezeigt werden, wie sich aus dem abstrakten Modell und den dafür entwickelten Algorithmen eine effiziente und skalierbare Implementierung entwickeln läßt. Es ergibt sich eine natürliche Korrespondenz zwischen baumförmigen Berechnungen und ihrer Implementierung durch einen abstrakten Datentyp im Rahmen einer portablen und wiederverwendbaren Bibliothek wie PIGSeL.

Das abstrakte Modell ist gut geeignet, den Lastverteilungsaspekt einer großen Klasse von Anwendungen zu modellieren. Manchmal genügt dies bereits, um verläßliche Voraussagen über die parallele Ausführungszeit zu machen. Bei anderen Anwendungen ist das Modell zwar nicht vollständig, aber dies hat keine großen Auswirkungen auf das Verhalten des parallelen Programmes. Selbst wenn das Verhalten nicht vollständig modelliert wird, bewähren sich die Lastverteilungsalgorithmen dennoch.

Ergebnisse

PIGSeL ist offenbar die erste Bibliothek für parallele Baumsuche, die die freie Kombination von Maschinen, Lastverteilern und Anwendungen erlaubt. Exponentielles zufälliges Anfragen mit zufälliger Initialisierung könnte durchaus der beste bisher implementierte Lastverteilungsalgorithmus für parallele Tiefensuche sein. Eine effizientes Durchsuchen unregelmäßiger Bäume auf Maschinen mit $\geq$ 1024 PEs war bisher jedenfalls nur für deutlich größere Probleme möglich. Die Implementierung des 0/1-Rucksackproblems scheint die erste sinn-

volle skalierbare Parallelisierung des Problems darzustellen und weist außerdem den Weg für verbesserte sequentielle Algorithmen. Weiter wurde gezeigt, daß die Parallelisierung von iterativem Vertiefen sich problemlos auf die Parallelisierung der Einzeliterationen zurückführen läßt, – vor allem wenn für die ersten Iterationen statische Lastverteilung eingesetzt wird.

Anhang B

Notation

Es wird weitgehend die übliche mathematische Notation verwendet. Die folgenden Vereinbarungen dienen deshalb hauptsächlich der Vorbeugung gegenüber möglichen Mißverständnissen.

Allgemeine mathematische Notation

$\mathbb{N}$, $\mathbb{N}_+$, $\mathbb{R}$, $\mathbb{R}_+$ **bzw.** $\mathbb{R}_*$ stehen für die natürlichen Zahlen (einschließlich der Null), die positiven ganzen Zahlen, die reellen Zahlen, die positiven reellen Zahlen respektive die nichtnegativen reellen Zahlen.

$[a,b]$, (a,b), $(a,b]$, $[a,b)$**:** Geschlossene, offene und halboffene reelle Intervalle.

$\mathbb{N}_n$**:** $\{0,\ldots,n-1\}$.

B^A**:** Menge aller Abbildungen von A nach B.

Bild f**:** Wertebereich einer Funktion.

$A \dot\cup B$**:** Disjunkte Vereinigung von A und B.

i,j,k,m,n**:** Natürliche Zahlen.

$\alpha,\beta,\gamma,\delta,\varepsilon,a,b,c,d$**:** Meist positive reelle Zahlen.

p,q**:** Wahrscheinlichkeiten.

$\forall,\exists$**:** Logische Quantoren. Der Bindungsbereich eines Quantors erstreckt sich bis zum Ende der Formel, wenn nicht durch Klammerung anders festgelegt. Wenn dies aus dem Kontext ersichtlich ist, werden Allquantoren z.T. weggelassen, es gilt also implizite Allquantifizierung.

$[F]$**:=** $\begin{cases} 1 & \text{falls das Prädikat } F \text{ wahr ist} \\ 0 & \text{sonst} \end{cases}$.

Diese Konvention stammt aus GRAHAM ET AL. (1992).

log**:** Logarithmus zur Basis 2.

$i \oplus j$**:** Bitweise exklusiv-oder Verknüpfung.

Asymptotische Notation

Definition B.1. In Anlehnung an BRASSARD UND BRATLEY (1988) sei für Funktionen $f \in {\mathbb{R}_*}^{\mathbb{N}}$ definiert:

$$\begin{aligned} \mathrm{O}(f) &:= \{g \in {\mathbb{R}_*}^{\mathbb{N}} \mid \exists c \in \mathbb{R}_+ : \exists n_0 : \forall n \geq n_0 : g(n) \leq cf(n)\} \\ \Omega(f) &:= \{g \in {\mathbb{R}_*}^{\mathbb{N}} \mid \exists c \in \mathbb{R}_+ : \exists n_0 : \forall n \geq n_0 : g(n) \geq cf(n)\} \\ \Theta(f) &:= \mathrm{O}(f) \cap \Omega(f) \end{aligned}$$

Analoge Definitionen lassen sich für Funktionen mit mehreren Parametern angeben. Arithmetische Operationen zwischen Funktionenmengen werden elementweise angewandt. Zum Beispiel steht $A+B$ für $\{a+b \mid a \in A \wedge b \in B\}$. Schreibweisen wie $f+A$ sind eine abkürzenden Schreibweise für $\{f\}+A$. Als leicht vom Üblichen abweichende Notation wird $f \preceq A$ als Abkürzung für $\exists g \in A : f \leq g$ benutzt.

Wenn nicht ausdrücklich anders erwähnt, werden asymptotische Analysen einer Größe immer für den schlechtesten Fall durchgeführt. Soll z.B. die Ausführungszeit $T(n)$ eines Algorithmus in Abhängigkeit von der Problemgröße betrachtet werden und gibt $t(P)$ die Ausführungszeit in Abhängigkeit von einer Probleminstanz an, so ist $T(n)$ als $\max\{t(P) \mid \text{Problemgröße}(P) = n\}$ definiert, ohne daß dies ausdrücklich gesagt werden muß.

Wahrscheinlichkeitsrechnung

Abgesehen von diversen Varianten in der Notation ist die grundlegende Begriffsbildung in der Wahrscheinlichkeitsrechnung zum Glück recht einheitlich. Als Grundlage dienen hier FELLER (1968) und HINDERER (1988).

- *Zufallsvariablen* sind Abbildungen von einem *Merkmalsraum* Ω auf einen Wertebereich. Jedem Element des Merkmalsraums ist eine Wahrscheinlichkeit zugeordnet. Meist haben alle Elemente des Merkmalsraums die gleiche Wahrscheinlichkeit. *0/1-Zufallsvariablen* oder *Indikatorzufallsvariablen* sind solche mit Wertebereich $\{0,1\}$.

- $\mathbf{P}[A]$ steht für die Wahrscheinlichkeit, bei einem Zufallsexperiment ein Element aus der durch das Prädikat A definierten Menge zu beobachten. $\mathbf{P}[A \mid B]$ ist die bedingte Wahrscheinlichkeit, ein Element aus A zu beobachten, falls ein Element aus B beobachtet wird.

- $\mathbf{E}X$ bezeichnet den Erwartungswert einer Zufallsvariablen X. Der bedingte Erwartungswert $\mathbf{E}[X \mid B]$ ist der Erwartungswert von X unter der Bedingung, daß ein Element aus B beobachtet wird. Die bedingte Erwartung $\mathbf{E}[X \mid Y]$ ist eine Zufallsvariable und zwar die Funktion $f(Y)$ mit $f(y) = \mathbf{E}[X \mid Y = y]$ (siehe auch MOTWANI UND RAGHAVAN (1995) Definition 4.4).

- $\mathbf{Var}X$ bezeichnet die Varianz einer Zufallsvariablen X.

Algorithmenbeschreibung

Algorithmen werden in einem Pascal- bzw. Algolähnlichen Pseudocode beschrieben, der ausgiebig mit mathematischer Notation angereichert ist, um eine präzisere und kompaktere Darstellung zu ermöglichen. Innerhalb des Pseudocodes wird ausschließlich Englisch verwendet, um unschön klingende Mischungen von deutschen Beschreibungen und englischen Schlüsselwörtern zu vermeiden. Da Kontrollstrukturen konsequent eingerückt sind, werden Schlüsselwörter zur Blockendemarkierung z.T. weggelassen um den Code nicht durch redundante Zeilen zu verwässern (siehe auch BRASSARD UND BRATLEY (1988)). Diese halbformale Darstellung wird durch begleitenden Text ergänzt, der u.a. weniger übliche Konstrukte erläutert.

Literaturverzeichnis

ACHILLES, A.-C., 1995a, A collection of computer science bibliographies. `http://liinwww/bibliography/index.html`.

ACHILLES, A.-C., 1995b, Optimal emulation of meshes on meshes-of-trees. In *EURO-PAR International Conference on Parallel Processing*, S. 193–204.

AHARONI, G., BARAK, A. UND FARBER, Y., 1993, An adaptive granularity control algorithm for the parallel execution of functional programs. *Future Generation Computing Systems*, **9**:163–174.

AKL, S. G., 1989, *The Design and Analysis of Parallel Algorithms*. Prentice-Hall.

ALON, N. UND SPENCER, J. H., 1992, *The Probabilistic Method*. Wiley.

ARVINDAM, S., KUMAR, V. UND RAO, V. N., 1990a, Efficient parallel algorithms for search problems: Applications in VLSI CAD. In *3rd Symposium on Massively Parallel Computation*.

ARVINDAM, S., KUMAR, V., RAO, V. N. UND SINGH, V., 1990b, Automatic test pattern generator on parallel processors. Tech. Ber. TR 90-20, University of Minnesota.

BECKER, W., 1995, Das HiCon-Modell. Dynamische Lastverteilung für datenintensive Anwendungen auf Rechnernetzen. *Informatik Forschung und Entwicklung*, **10**:14–25.

BLOOM, G. S. UND GOLOMB, S. W., 1977, Applications of numbered undirected graphs. *Proceedings of the IEEE*, **65**(4):562–570.

BLUMOFE, R. D., JOERG, C., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H. UND ZHOU, Y., 1995, Cilk: An efficient multithreaded runtime system. In *Principles and Practice of Parallel Programming*, S. 207–216.

BLUMOFE, R. D. UND LEISERSON, C. E., 1994, Scheduling multithreaded computations by work stealing. In *Foundations of Computer Science*, S. 356–368, Santa Fe.

BÖHM, M. UND SPECKENMEYER, E., 1994, Effiziente Lastausgleichsverfahren. In *Transputer Anwender Treffen*, S. 53–60, Aachen, IOS Press.

BRASSARD, G. UND BRATLEY, P., 1988, *Algorithmics – Theory and Practice*. Prentice Hall.

BRIDGES, P., DOSS, N., GROPP, W., KARRELS, E., LUSK, E. UND SKJELLUM, A., 1995, *User's Guide to* `mpich`*, a Portable Implementation of MPI*. Argonne National Laboratory.

BUTENUTH, R., BURKE, W. UND HEISS, H.-U., 1996, COSY – an operating system for highly parallel computers. *ACM Operating Systems Review*, **30**(2):81–91.

CASARI, C., CATELLI, C., GUANZIROLI, M., MAZZEO, M., MEOLA, G., PUNZI, S., SCHEININE, A. UND STOFELLA, P., 1994, Status report on ESPRIT project p9519 palace: Parallelization of GEANT. In *International Conference Massively Parallel Processing Applications and Development*, S. 405–412, Delft, Elsevier.

CHAKRABARTI, S., RANADE, A. UND YELICK, K., 1994, Randomized load balancing for tree-structured computation. In *Scalable High Performance Computing Conference*, S. 666–673, Knoxville.

CHERNOFF, H., 1952, A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, **23**:493–507.

CHONG, F. T., BREWER, E. A., LEIGHTON, F. T. UND KNIGHT JR., T. F., 1994, Building a better butterfly: The multiplexed metabutterfly. In *International Symposium on Parallel Architectures Algorithms and Networks*, Kanazawa, Japan.

CHRISTMAN, D. P., 1983, *Programming the Connection Machine*. Diplomarbeit, MIT.

CLAUS, V. UND SCHWILL, A., 1988, *Duden Informatik*. Duden Verlag.

CZUMAJ, A., KNAREK, P., KUTYLOWSKI, M. UND LORYS, K., 1996, Fast generation of random permutations via network simulation. In *European Symposium on Algorithms*.

DE DONCKER, E., EALY, P. UND GUPTA, A., 1996, Two methods for load balanced distributed adaptive integration. In *HPCN Europe*, S. 562–570.

DEHNE, F., FERREIRA, A. G. UND RAU-CHAPLIN, A., 1990a, Parallel AI algorithms for fine-grained hypercube multiprocessors. In *Parcella 90, V. International Workshop on Parallel Proccessing by Cellular Automata and Arrays*, S. 51–65, Berlin.

DEHNE, F., FERREIRA, A. G. UND RAU-CHAPLIN, A., 1990b, Parallel branch and bound on fine-grained hypercube multiprocessors. *Parallel Computing*, **15**:201–209.

DIEFENBACH, P. UND SANDERS, P., 1995, *PIGSel Manual*.

DION, M., GENGLER, M. UND UBEDA, S., 1994, Comparing two probabilistic models of the computational complexity of the branch and bound algorithm. In *CONPAR/VAPP*, S. 359–370, Linz, Springer.

DOLLAS, A., RANKIN, W. T. UND MCCRACKEN, D., 1995, New algorithms for golomb ruler derivation and proof of the 19 mark ruler. Tech. Ber., Duke University, North Carolina, `http://www.ee.duke.edu/~wrankin/golomb`.

DUTT, S. UND MAHAPATRA, M. R., 1993, Parallel A* algorithms and their performance on hypercube multiprocessors. In *International Parallel Processing Symposium*, Newport Beach.

EL-DESSOUKI, O. I. UND HUEN, W. H., 1980, Distributed enumeration on between computers. *IEEE Transactions on Computers*, **C-29**(9):818–825.

ERTEL, W., 1992, *Parallele Suche mit randomisiertem Wettbewerb in Inferenzsystemen*. Dissertation, TU München.

FELDMANN, R., 1993, *Game Tree Search on Massively Parallel Systems*. Dissertation, Universität Paderborn.

FELDMANN, R., MYSLIWIETZ, P. UND MONIEN, B., 1991, Distributed game tree search on a massively parallel system. In *Data structures and efficient algorithms: Final report on the DFG special joint initiative*, Nr. 594 in LNCS, S. 270–288.

FELDMANN, R., MYSLIWIETZ, P. UND MONIEN, B., 1994, Studying overheads in massively parallel min/max-tree evaluation. In *ACM Symposium on Parallel Architectures and Algorithms*, S. 94–103.

FELLER, W., 1968, *An Introduction to Probability Theory and its Applications*. Wiley, 3. Aufl.

FELTEN, E. W., 1988, Best-first branch-and-bound on a hypercube. In *Proceedings of the 3rd Conference on Hypercubes, Concurrent Computers and Applications*, S. 1500–1504, Pasadena.

FINKEL, R. UND MANBER, U., 1987, DIB – A distributed implementation of backtracking. *ACM Trans. Prog. Lang. and Syst.*, **9**(2):235–256.

FRYE, R. UND MYCZKOWSKI, J., 1991, Exhaustive search of unstructured trees on the Connection Machine. Tech. Ber. 196, Thinking Machines Corporation.

FUNKE, R., LÜLING, R., MONIEN, B., LÜCKING, F. UND BLANKE-BOHNE, H., 1992, An optimized reconfigurable architecture for transputer networks. In *25th Hawaii Int. Conference on System Sciences*, Bd. 1, S. 237–245, Hawaii.

GASSER, R. U., 1995, *Harnessing Computational Resources for Efficient Exhaustive Search*. Dissertation, ETH Zürich.

GEIST, A. ET AL., 1993, *PVM 3.0 Users's Guide and Reference Manual*. Oak Ridge National Laboratory, ORNL/TM-12187.

GHOSH, B., LEIGHTON, F. T., MAGGS, B. M., MUTHUKRISHNAN, S., PLAXTON, C. G., RAJARAMAN, R., RICHA, A. W., TARJAN, T. E. UND ZUCKERMAN, D., 1995, Tight analyses of two local load balancing algorithms. In *ACM Symposium on the Theory of Computing*.

GIBBONS, P. B., MATIAS, Y. UND RAMACHANDRAN, V., 1994, Efficient low contention parallel algorithms. In *ACM Symposium on Parallel Architectures and Algorithms*, S. 236–247, Cape May, NJ.

GMEHLICH, R., 1994, *Einbettungen von Graphen mit logarithmischem Durchmesser in Gitter*. Diplomarbeit, Universität Karlsruhe.

GRAHAM, R. L., KNUTH, D. E. UND PATASHNIK, O., 1992, *Concrete Mathematics*. Addison Wesley.

GUPTA, R., SMOLKA, S. A. UND BHASKAR, S., 1994, On randomization in sequential and distributed algorithms. *ACM Computing Surveys*, **26**(1):7–86.

HAGERUP, T., 1991, Fast parallel generation of random permutations. In *International Colloquium on Automata, Languages and Programming*.

HEIDELBERGER, P., 1988, Discrete event simulations and parallel processing: Statistical properties. *SIAM Journal of Statistical Computation*, **9**(6):1114–1132.

HELMAN, P., 1988, An algebra for search problems and their solutions. In *Search in Artificial Intelligence*, herausgegeben von V. Kumar und L. N. Kanal, S. 28–90, Springer.

HENNECKE, M., 1994, Parallelisierung von Zufallszahlengeneratoren. In *Drittes ODIN Symposium*, S. 143–150, Karlsruhe.

HENRICH, D., 1994, *Lastverteilung für Branch-and-bound auf eng-gekoppelten Parallelrechnern*. Dissertation, Universität Karlsruhe.

HILLIS, W. D., 1985, *The Connection Machine*. Series in Artificial Intelligence, MIT Press, Cambridge, MA.

HILLIS, W. D. UND TUCKER, L. W., 1993, The CM-5 connection machine: A scalable supercomputer. *Communications of the ACM*, **36**(11):30–40.

HINDERER, K., 1988, Grundlagen der Stochastik für Informatiker und Ingenieure. Vorlesungsskript, Universität Karlsruhe.

HOPCROFT, J. E. UND ULLMAN, J. D., 1979, *Introduction to Automata Theory, Languages, and computation*. Addison-Wesley.

HOPP, H., 1995, *Parallele Spielbaumsuche auf SIMD-Rechnern*. Diplomarbeit, Universität Karlsruhe.

HOPP, H. UND SANDERS, P., 1995, Parallel game tree search on SIMD machines. In *Workshop on Algorithms for Irregularly Structured Problems*, Nr. 980 in LNCS, S. 349–361, Lyon, Springer.

HOROWITZ, E. UND SAHNI, S., 1974, Computing partitions with applications to the knapsack problem. *Journal of the ACM*, **21**(2):277–292.

HWANG, K., 1993, *Advanced Computer Architecture – Parallelism Scalability Programmability*. McGraw Hill.

IERARDI, D., 1994, 2d-bubblesorting in average time $O(\sqrt{N \lg N})$. In *ACM Symposium on Parallel Architectures and Algorithms*.

JANAKIRAM, V. K., AGRAWAL, D. P. UND MEHOTRA, R., 1987, Randomized parallel algorithms for Prolog programs and backtracking applications. In *International Conference on Parallel Processing*, S. 278–281.

JANAKIRAM, V. K., GEHRINGER, E. F., AGRAWAL, D. P. UND MEHOTRA, R., 1988, A randomized parallel branch-and-bound algorithm. *International Journal of Parallel Programming*, **17**(3):277–301.

JOHNSON, N. L., KOTZ, S. UND KEMP, A. W., 1989, *Univariate Discrete Distributions*. Wiley.

KAKLAMANIS, C. UND PERSIANO, G., 1994, Branch-and-bound and backtrack search on mesh-connected arrays of processors. *Mathematical Systems Theory*, **27**:471–489.

KALE, L. V. UND SINHA, A. B., 1991, Information sharing mechanisms in parallel programs. Tech. Ber., University of Illinois, Urbana Champaign.

KARP, R. M., 1983, Searching for an optimal path in a tree with random costs. *Artificial Intelligence*, **21**:99–116.

KARP, R. M. UND ZHANG, Y., 1993, Parallel algorithms for backtrack search and branch-and-bound. *Journal of the ACM*, **40**(3):765–789.

KARYPIS, G. UND KUMAR, V., 1992, Unstructured tree search on SIMD parallel computers. Tech. Ber., University of Minnesota.

KARYPIS, G. UND KUMAR, V., 1994, Unstructured tree search on SIMD parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, **5**(10):1057–1072.

KERGOMMEAUX, J. C. UND CODOGNET, P., 1994, Parallel logic programming systems. *ACM Computing Surveys*, **26**(3):295–336.

KHAMBEKAR, P., 1992, *Dynamic Load Balancing in Distributed Memory Systems*. Ph.d. thesis, Clemson University.

KNUTH, D. E., 1981, *The Art of Computer Programming — Seminumerical Algorithms*, Bd. 2. Addison Wesley, 2. Aufl.

KORF, R. E., 1985, Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, **27**:97–109.

KRUSKAL, C. P. UND WEISS, A., 1985, Allocating independent subtasks on parallel processors. *IEEE Transactions on Computers*, **11**(10):1001–1016.

KUMAR, V. UND ANANTH, G. Y., 1991, Scalable load balancing techniques for parallel computers. Tech. Ber. TR 91-55, University of Minnesota.

KUMAR, V., GRAMA, A., GUPTA, A. UND KARYPIS, G., 1994, *Introduction to Parallel Computing. Design and Analysis of Algorithms*. Benjamin/Cummings.

KUMAR, V. UND RAO, V. N., 1990, Scalable parallel formulations of depth-first search. In *Parallel Algorithms for Machine Intelligence and Vision*, herausgegeben von V. Kumar, Springer.

LAUER, T., 1995, *Adaptive dynamische Lastbalancierung*. Dissertation, MPI für Informatik Saarbrücken.

L'ECUYER, P., 1990, Random numbers for simulation. *Communications of the ACM*, **33**(10):85–97.

LEIGHTON, F. T., MAGGS, B. M., RANADE, A. G. UND RAO, S. B., 1994, Randomized routing and sorting on fixed-connection networks. *Journal of Algorithms*, **17**:157–205.

LEIGHTON, T., 1992, *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann.

LEIGHTON, T., NEWMAN, M., RANADE, A. G. UND SCHWABE, E., 1989, Dynamic tree embeddings in butterflies and hypercubes. In *ACM Symposium on Parallel Architectures and Algorithms*, S. 224–234.

LEISERSON, C. E., 1985, Fat trees: Universal networks for hardware efficient supercomputing. In *International Conference on Parallel Processing*, S. 393–402.

LIN, W. UND YANG, B., 1995, Probabilistic performance analysis for parallel search techniques. *International Journal of Parallel Programming*, **23**(2):161–189.

LIN, Z., 1991, A distributed fair polling scheme applied to OR-parallel Prolog. *International Journal of Parallel Programming*, **20**(4):315–339.

LOOTS, W. UND SMITH, T. H. C., 1992, A parallel algorithm for the 0-1 knapsack problem. *International Journal of Parallel Programming*, **21**(5):349–362.

LÜLING, R. UND MONIEN, B., 1992, Load balancing for distributed branch & bound algorithms. In *Int. Parallel Processing Symposium (IPPS)*.

LÜLING, R., MONIEN, B. UND RAMME, F., 1991, Load balancing in large networks: A comparative case study. In *3th IEEE Symposium on Parallel and Distributed Processing*, IEEE.

MA, R. P., TSUNG, F. S. UND MA, M. H., 1988, A dynamic load balancer for a parallel branch and bound algorithm. In *3rd Conference on Hypercubes, Concurrent Computers, and Applications*, S. 1505–1530, Pasadena, ACM.

MARTELLO, S. UND TOTH, P., 1990, *Knapsack Problems – Algorithms and Computer Implementations*. Wiley.

MasPar, 1992, *MasPar Programming Language User Guide*. MasPar Computer Corporation.

MATTEIS, A. D. UND PAGNUTTI, S., 1990, A class of parallel random number generators. *Parallel Computing*, **13**:193–198.

MATTERN, F., 1987, *Verteilte Basisalgorithmen.* Nr. 226 in Informatik-Fachberichte, Springer.

MAYR, E. W. UND WERCHNER, R., 1993, Divide-and-conquer algorithms on the hypercube. Tech. Ber. TUM-I9322, TU München.

MCKEOWN, G. P., RAYWARD-SMITH, V. J. UND RUSH, S. A., 1992, Parallel branch-and-bound. In *Advances in Parallel Algorithms*, S. 349–362, Blackwell.

MEIKO, 1993a, Computing surface communications network overview. Tech. Ber.

MEIKO, 1993b, Computing surface communications processor overview. Tech. Ber.

MEYER AUF DER HEIDE, F., OESTERDIEKHOFF, B. UND WANKA, R., 1993, Strongly adaptive token distribution. In *ICALP*, S. 398–409.

MONIEN, B., FELDMANN, R., MYSLIWIETZ, P. UND VORNBERGER, O., 1990, Parrallel game tree search by dynamic tree decomposition. In *Parallel Algorithms for Machine Intelligence and Vision*, herausgegeben von V. Kumar, P. S. Gopalakrishnan und L. Kanal, Springer-Verlag, New York.

Motorola, 1993, *PowerPC Optimizing C Compilation System User's Guide*. Motorola.

MOTWANI, J. UND RAGHAVAN, P., 1995, *Randomized Algorithms*. Cambridge University Press.

MPI FORUM, 1996, MPI-2: Extensions to the message-passing interface standard. Tech. Ber., University of Tennessee.

MÜLLER, P. H., 1991, *Lexikon der Stochastik.* Akademie Verlag, 5. Aufl.

NAGANUMA, J., 1993, A highly OR-parallel inference machine (multi-ACSA) and its performance evaluation: An architecture and its load balancing algorithm. *IEEE Transactions on Computers*, **43**(9):1062–1075.

NATARAJAN, K. S., 1989, Expected performance of parallel search. In *International Conference on Parallel Processing*, Bd. III, S. 121–125.

NAU, D. S., KUMAR, V. UND KANAL, L., 1984, General branch and bound, and its relation to A* and AO*. *Artificial Intelligence*, **23**:29–58.

NICOL, D. M. UND SALTZ, J. H., 1990, An analysis of scatter decomposition. *IEEE Transactions on Computers*, **39**:1337–1345.

NIEDERMEIER, R. UND SANDERS, P., 1996, On the Manhattan-distance between points on space-filling mesh-indexings. Tech. Ber. IB 18/96, Universität Karlsruhe, Fakultät für Informatik.

NONNENMACHER, A. UND MLYNSKI, D. A., 1996, Liquid crystal simulation using automatic differentiation and interval arithmetic. In *Scientific Computing and Validated Numerics*, herausgegeben von G. Alefeld und A. Frommer, Akademie Verlag.

OED, W., 1993, The Cray research massively parallel processor system Cray T3D. Tech. Ber., Cray Research.

PEARL, J., 1984, *Heuristics*. Addison Wesley.

PELEG, D. UND UPFAL, E., 1989, The token distribution problem. *SIAM Journal of Computing*, **18**:229–243.

PHILIPPSEN, M., WARSCHKO, T., TICHY, W. F. UND HERTER, C., 1992, Projekt Triton: Beiträge zur Verbesserung der Programmierbarkeit hochparalleler Rechensysteme. *Informatik Forschung und Entwicklung*, S. 1–13.

POWLEY, C., FERGUSON, C. UND KORF, R. E., 1993, Depth-first heuristic search on a SIMD machine. *Artificial Intelligence*, **60**:199–242.

PRECHELT, L., 1993a, Comparison of MasPar MP-1 and MP-2 communication operations. Tech. Ber. IB 16/93, Universität Karlsruhe.

PRECHELT, L., 1993b, Measurements of MasPar MP-1216a communication operations. Tech. Ber. IB 1/93, Universität Karlsruhe.

PRESS, W. H., TEUKOLSKY, S., VETTERLING, W. T. UND FLANNERY, B. P., 1992, *Numerical Recipes in C*. Cambridge University Press, 2. Aufl.

RAJASEKARAN, S., 1992, Randomized algorithms for packet routing on the mesh. In *Advances in Parallel Algorithms*, herausgegeben von L. Kronsjö und D. Shumsheruddin, S. 277–301, Blackwell.

RANADE, A., 1994, Optimal speedup for backtrack search on a butterfly network. *Mathematical Systems Theory*, S. 85–101.

RAO, V. N. UND KUMAR, V., 1987a, Parallel depth first search. Part I. *International Journal of Parallel Programming*, **16**(6):470–499.

RAO, V. N. UND KUMAR, V., 1987b, Parallel depth first search. Part II. *International Journal of Parallel Programming*, **16**(6):501–519.

RAO, V. N. UND KUMAR, V., 1993, On the efficiency of parallel backtracking. *IEEE Transactions on Parallel and Distributed Systems*, **4**(4):427–437.

RATNER, D. UND WARMUTH, M., 1990, Finding a shortest solution to the $n \times n$ extension of the 15-puzzle is intractable. *Journal of Symbolic Computation*, **10**:111–137.

REIF, J. H., 1985, An optimal parallel algorithm for integer sorting. In *Foundations of Computer Science*, S. 496–504.

REIF, J. H. UND SEN, S., 1994, Randomized algorithm for binary search and load balancing on fixed connection networks with geometric applications. *SIAM Journal on Computing*, **23**(3):633–651.

REINEFELD, A., 1994, Scalability of massively parallel depth-first search. In *DIMACS Workshop*.

REINEFELD, A. UND SCHNECKE, V., 1994, AIDA* — asynchronous parallel IDA*. In *10th Canadian Conference on Artificial Intelligence*, Banff.

SANDERS, P., 1993, *Suchalgorithmen auf SIMD-Rechnern – Weitere Ergebnisse zu Polyautomaten*. Diplomarbeit, Universität Karlsruhe.

SANDERS, P., 1994a, Analysis of random polling dynamic load balancing. Tech. Ber. IB 12/94, Universität Karlsruhe, Fakultät für Informatik.

SANDERS, P., 1994b, A detailed analysis of random polling dynamic load balancing. In *International Symposium on Parallel Architectures, Algorithms and Networks*, S. 382–389, Kanazawa, Japan.

SANDERS, P., 1994c, Emulating MIMD behavior on SIMD machines. In *International Conference Massively Parallel Processing Applications and Development*, S. 313–321, Delft, Elsevier.

SANDERS, P., 1994d, Massively parallel search for transition-tables of polyautomata. In *Parcella 94, VI. International Workshop on Parallel Proccessing by Cellular Automata and Arrays*, S. 99–108, Potsdam.

SANDERS, P., 1994e, Portable parallele Baumsuchverfahren: Entwurf einer effizienten Bibliothek. In *Transputer Anwender Treffen*, S. 168–177, Aachen, IOS Press.

SANDERS, P., 1994f, Randomized static load balancing for tree shaped computations. In *Workshop on Parallel Processing*, TR Universität Clausthal, S. 58–69, Lessach, Österreich, ISSN 0943-3805.

SANDERS, P., 1995a, Better algorithms for parallel backtracking. In *Workshop on Algorithms for Irregularly Structured Problems*, Nr. 980 in LNCS, S. 333–347, Lyon, Springer.

SANDERS, P., 1995b, Efficient emulation of MIMD behavior on SIMD machines. Tech. Ber. IB 29/95, Universität Karlsruhe, Fakultät für Informatik.

SANDERS, P., 1995c, Fast priority queues for parallel branch-and-bound. In *Workshop on Algorithms for Irregularly Structured Problems*, Nr. 980 in LNCS, S. 379–393, Lyon, Springer.

SANDERS, P., 1995d, Some implementation results on random polling dynamic load balancing. Tech. Ber. 40/95, Universität Karlsruhe.

SANDERS, P., 1995e, Towards better algorithms for parallel backtracking. Tech. Ber. IB 6/95, Universität Karlsruhe, Fakultät für Informatik.

SANDERS, P., 1996a, On the competitive analysis of randomized static load balancing. In *First Workshop on Randomized Parallel Algorithms*, herausgegeben von S. Rajasekaran, Honolulu, Hawaii.

SANDERS, P., 1996b, On the efficiency of nearest neighbor load balancing for random loads. In *Parcella 96, VII. International Workshop on Parallel Proccessing by Cellular Automata and Arrays*, S. 120–127, Berlin, Akademie Verlag.

SANDERS, P., 1996c, Optimizing the emulation of MIMD behavior on SIMD machines. In *Parcella 96, VII. International Workshop on Parallel Proccessing by Cellular Automata and Arrays*, Akademie Verlag.

SANDERS, P., 1996d, A scalable parallel tree search library. In *2nd Workshop on Solving Irregular Problems on Distributed Memory Machines*, herausgegeben von S. Ranka, Honolulu, Hawaii.

SANDERS, P. UND WORSCH, T., 1995, Konvergente lokale Lastverteilungsverfahren und ihre Modellierung durch Zellularautomaten. In *PARS Workshop*, Bd. 14 von *PARS Mitteilungen*, S. 285–291.

SHAVIT, N. UND ZEMACH, A., 1994, Diffracting trees. In *ACM Symposium on Parallel Architectures and Algorithms*, S. 167–176, Cape May, New Jersey.

SNIR, M., HOCHSCHILD, P., FRYE, D. D. UND GILDEA, K. J., 1995, The communication software and parallel environment of the IBM SP-2. *IBM Systems Journal*, **34**(2).

SNIR, M., OTTO, S. W., HUSS-LEDERMAN, S., WALKER, D. W. UND DONGARRA, J., 1996, *MPI the Complete Reference*. MIT Press.

TSCHÖKE, S., RÄCKE, M., LÜLING, R. UND MONIEN, B., 1994, Solving the traveling salesman problem with a parallel branch-and-bound algorithm on a 1024 processor network. Tech. Ber., Universität Paderborn.

ULLMAN, J. D., 1984, *Computational Aspects of VLSI*. Computer Science Press.

UMLAND, T. UND VOLLMAR, R., 1992, *Transputerpraktikum*. Teubner.

UPFAL, E., 1994, On the theory of interconnection networks for parallel computers. In *Scalable High Performance Computing Conference*, S. 473–486, Knoxville.

VALIANT, L., 1994, A bridging model for parallel computation. *Communications of the ACM*, **33**(8).

VOLLMAR, R., 1979, *Algorithmen in Zellularautomaten*. Teubner.

VOLLMAR, R. UND WORSCH, T., 1995, *Modelle der Parallelverarbeitung*. Teubner.

VORNBERGER, O., 1987, Implementing branch-and-bound in a ring of processors. In *CONPAR*, S. 157–164, Springer.

WORSCH, T., 1994, Lower and upper bounds for (sums of) binomial coefficients. Tech. Ber. IB 31/94, Universität Karlsruhe.

WU, I. C. UND KUNG, H. T., 1991, Communication complexity of parallel divide-and-conquer. In *Foundations of Computer Science*, S. 151–162.

YU, X. UND GHOSAL, D., 1992, Optimal dynamic tree scheduling of task tree on constant-dimensional architectures. In *ACM Symposium on Parallel Architectures and Algorithms*, S. 138–146.

Index

Symbole

A

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z