Lastverteilung für feinkörnig parallelisiertes Branch-and-bound

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

der Fakultät für Informatik

der Universität Karlsruhe (Technische Hochschule)

vorgelegte Dissertation von

Dominik Henrich

aus Ulm an der Donau

Tag der mündlichen Prüfung:	10. Januar 1995
Hauptreferent:	Prof. Dr.-Ing. U. Rembold
Korreferent:	Prof. Dr.-Ing. R. Vollmar

Geleitwort

Ein klassisches Problem der Künstlichen Intelligenz ist die effiziente Abarbeitung von Suchbäumen. Zu den damit gegebenen Aufgabenstellungen gehören u.a. die Suche nach kürzesten Wegen, die Lösung des Constraint Satisfaction Problem oder die hier betrachtete Lösung der diskreten Optimierung. Für die Optimierung wird oft das aus dem Operations Research bekannte Baumsuchverfahren "Branch-and-bound" verwendet. In der Künstlichen Intelligenz ist es unter dem Namen A* bekannt.

Typischerweise sind die diskreten Optimierungsprobleme, wie z.B. das Problem des Handlungsreisenden, das Rucksackproblem oder die Maschinenbelegungsplanung, von hoher Komplexität und somit durch lange Berechnungszeiten gekennzeichnet. Da die technische Entwicklung sequentieller Rechner zunehmend an physikalische Grenzen stößt, ist die Verwendung massiver Rechenparallelität eine vielversprechende Alternative. Bei Parallelrechnern muß aber eine ausgeglichene Beschäftigung der Prozessoren zur Aufrechterhaltung der Effizienz gewährleistet sein. Gerade für feinkörnige Parallelrechner mit mehreren 10.000 Prozessorelementen stellt die dazu erforderliche Lastverteilung der anstehenden Rechenarbeit auf die Prozessoren ein zentrales Problem dar. In diesem Buch wird die interessante Aufgabe der Lastverteilung für ein verbreitetes Baumsuchverfahren angegangen.

Das systematische Verfolgen der möglichen Ansatzpunkte zur Lastverteilung bei paralleler Baumsuche resultiert in drei Ansätzen, die sich aus den verschiedenen Einsatzzeitpunkten der Lastverteilung ergeben. Für jeden dieser Ansatzpunkte werden eigenständige Methoden beschrieben. Es gibt für die Versorgung der Prozessoren mit Suchbaumknoten vor der eigentlichen Suche ein Reihe von Verfahren, die hier erstmalig als solche herausgestellt und miteinander verglichen werden. Es wird durch ein einfaches physikalisches Flüssigkeitsmodell die enge Kopplung der Prozessoren zur effizienten Lastverteilung während der Baumsuche ausgenutzt. Alternativ dazu kann die Lastverteilung mit dem Branch-and-bound-Prozeß verschmolzen werden, wobei ein neuartiger Grundmechanismus zur Ausführung der parallelen Baumsuche eingeführt wird.

In diesem Buch wird gezeigt, daß die massiv-parallele Ausführung der stark dynamischen Baumsuch-Algorithmen auf feinkörnigen SIMD-Architekturen bei entsprechender Initialisierung der Prozessoren auch mit strikt lokalen Lastverteilungsmethoden effizient möglich ist. Dies stellt einen weiteren Schritt zur Anwendung der aufwendigen Optimierungsalgorithmen für reale Problemstellungen dar. Es handelt sich um eine grundlegende Arbeit, die ein klassisches Forschungsgebiet der Künstlichen Intelligenz mit der zukunftsweisenden Methode der massiv-parallelen Datenverarbeitung kombiniert und einen wesentlichen Beitrag zur Lösung diskreter Optimierungsprobleme leistet.

Prof. Dr.-Ing. U. Rembold

Danksagung

Die vorliegende Arbeit entstand im Rahmen des Graduiertenkollegs "Beherrschbarkeit komplexer Systeme" der Fakultät für Informatik, Universität Karlsruhe und wurde durch ein dreijähriges Stipendium der Deutschen Forschungsgemeinschaft (DFG) gefördert.

Ganz herzlich bedanke ich mich bei meinem Doktorvater Herrn Prof. Dr.-Ing. U. Rembold für seine erfahrene Unterstützung meiner Arbeit sowie für die Integration am Institut für Prozeßrechentechnik und Robotik.

Bei Herrn Prof. Dr.-Ing. R. Vollmar bedanke ich mich sehr für die Übernahme des Korreferats und für seine Betreuung, auch im Rahmen seiner Tätigkeit als Sprecher des Graduiertenkollegs.

Außerdem möchte ich mich bei allen Kollegen am Institut für ihre Kooperationsbereitschaft bedanken. Besonderer Dank gilt:

- Dipl.-Inform. Peter Sanders für die vielen fachlichen Auseinandersetzungen über Lastverteilung und parallele Baumsuche,
- meinem Zimmerkollegen Dipl.-Inform. Walter Reithofer, welcher immer zum Diskutieren eines noch so ausgefallenen Themas bereit war und diese Arbeit kritisch durchgesehen hat,
- Dipl.-Inform. Dipl.-Ing. Björn Magnussen für die Durchsicht der Arbeit aus dem Blickwinkel eines Ingenieurs.

Zudem bedanke ich mich bei "meinen" Studenten Bernd Augustin, Jeannette Hassler, Jörg Hecker, Jürgen Hemminger, Mario Sailer sowie Martin Stalp für die vielen nützlichen Diskussionen um das eine oder andere Detail.

Schließlich möchte ich meiner Frau Uta Birkner für ihre seelische Unterstützung in jeder Lebenslage danken.

Karlsruhe, im Januar 1995 Dominik Henrich.

Kurzfassung

Zur Planung und Steuerung von komplexen rechnerintegrierten Fertigungssystemen (CIM) ist die Abarbeitung vieler extrem aufwendiger Algorithmen notwendig. Aus dem Bereich der Fertigungssteuerung zählt die Generierung von Maschinenbelegungsplänen (scheduling) dazu. Zur Steigerung der Lösungsgeschwindigkeit bzw. zum Erreichen exakter Ergebnisse bietet sich der massive Einsatz von Rechenparallelität an. Mit Parallelrechnern ist durch die gleichzeitige Verwendung von vielen Prozessoren potentiell eine sehr große Leistungssteigerung zu erreichen. Dafür muß jedoch die vorhandene Parallelität effektiv genutzt werden. Die dazu erforderliche Verteilung der anstehenden Arbeit auf eine große Menge von Prozessoren heißt *Lastverteilung* und stellt den Kern dieser Arbeit dar.

Als allgemeiner Algorithmus zur Lösung kombinatorischer Optimierungsprobleme wird das Branch-and-bound-Verfahren eingesetzt und auf feinkörnigen Parallelrechnerarchitekturen ausgeführt. Zur Lastverteilung werden folgende drei Ansätze verfolgt und untersucht:

- *Statische Lastverteilung*: Es werden mehrere Methoden zur Initialisierung der Prozessoren, welche vor dem eigentlichen Optimierungsalgorithmus ausgeführt werden, analysiert. Es zeigt sich, daß sich die statische Lastverteilung überproportional stark auf die Laufzeit des nachfolgenden Branch-and-bound-Algorithmus auswirkt. Es ist daher wichtig, der bisher unterschätzten statischen Lastverteilung für die parallele Baumsuche mit realen Problemstellungen, besondere Aufmerksamkeit zu schenken.
- *Dynamische Lastverteilung*: Es wird ein vereinfachtes, gut skalierbares Flüssigkeitsmodell als erste synchrone lokale Lastverteilung entwickelt, welche besonders für Parallelrechner mit kurzer Verzögerungszeit beim Aufbau von Kommunikationsverbindungen effizient ist. Die Methode wird mit dem bekannten, aus dem Asynchronen übertragenen, Mittelungs-Ansatz verglichen. Zum analytischen Vergleich wird als ein realistischeres Aufwandsmaß die Kommunikationsmenge statt der üblichen Anzahl von Kommunikationsschritte verwendet. Der in der Prozessoranzahl bisher benötigte quadratische Zeitaufwand wird durch das Flüssigkeitsmodell auf einen linearen Aufwand reduziert, wobei das Flüssigkeitsmodell auch bzgl. der konstanten Zeitfaktoren signifikant effizienter ist.
- *Implizite Lastverteilung*: Zur Vermeidung von Wartezeiten der unbeteiligten Prozessoren während der Lastverteilung wird der Lastverteilungsprozeß mit dem Branch-and-bound-Prozeß verschmolzen. Das neuartige Konzept der k-Expansion unterstützt eine automatische Lastverteilung und approximiert eine globale Suchstrategie.

Zur Validierung der Ergebnisse werden Simulationen und Experimente mit einem Satz von Benchmark-Problemen durchgeführt. Der zugrunde liegende SIMD-Rechner ist eine MasPar MP-1 mit 16.384 Prozessoren in einem 2-dimensionalen Torus. Als exemplarische, NP-harte Anwendungsdomäne werden statische, non-operationale Planungsprobleme betrachtet.

Inhaltsverzeichnis

1. Einleitung

1.1 Motivation

In einem Fertigungsunternehmen sind Menschen und Maschinen mit Hilfe der Datenverarbeitung zu einem komplexen System verknüpft. Die Beherrschbarkeit dieses komplexen Systems hinsichtlich seiner Planung, Konfigurierung und Steuerung stellt ein wichtiges Ziel der Informationsverarbeitung dar. Um flexibel und schnell auf Anforderungen des Marktes reagieren zu können, müssen Produktionsanlagen oft äußerst effizient auf neue Produktwünsche abgestimmt werden. Nur mit Konzepten, welche den Produktionsprozeß von der Planung der Fabrik bis zur Fertigung des Endprodukts unterstützen, kann dieser Herausforderung erfolgreich begegnet werden. Computer integrated manufacturing (CIM) stellt einen allgemein anerkannten Ansatz zur Lösung dieses Problems dar [Rembold93].

In diesem Zusammenhang werden viele unterschiedliche und aufwendige Algorithmen verwendet. Aus dem Bereich der Fertigungsplanung und -steuerung zählt die Generierung von Maschinenbelegungsplänen (scheduling) dazu. Hierbei kann durch eine optimale Zuordnung der anstehenden Aufträge auf die verfügbaren Maschinen eine entscheidende Kostenreduktion erzielt werden. Zusätzlich erfordert die on-line Anbindung der Fertigungsplanung an den technischen Prozeß Planungsalgorithmen, die auf sich ändernde Situationen, wie z.B. Maschinenausfälle oder Eilaufträge, innerhalb vorgegebener Zeitschranken reagieren können.

Die Problematik der Maschinenbelegungsplanung ist gekennzeichnet durch eine hohe Komplexität, vor allem für die Berechnung von optimalen Plänen. Bei den dafür bekannten Algorithmen steigt der Lösungsaufwand exponentiell mit der Problemgröße an. Mit den heutigen Rechenleistungen sind realistische Aufgaben der Belegungsplanung, wie sie schon in mittelständischen Unternehmen vorkommen, nicht in vertretbarer Zeit zu handhaben. Schnellere Berechnungsmethoden sind meistens notwendig, denn nicht zu vermeidende

Störungen erfordern eine partielle Neuberechnung innerhalb vorgegebener Zeitschranken.

Sequentielle Rechner sind heute nur noch durch großen technologischen Aufwand zu beschleunigen, da man sich mit der Leistungsfähigkeit den physikalischen Grenzen nähert. Sie scheiden daher für einen längerfristigen Ansatz aus. Als Ausweg bietet sich der Einsatz von massiver Parallelität an. Mit Parallelrechnern ist durch Hinzunahme von weiteren Prozessoren und mit *entsprechend komplexen* Problemen potentiell eine beliebig große Leistungssteigerung zu erreichen.[1] Dafür muß das Problem bewältigt werden, die vorhandene Parallelität effektiv auszunutzen. Lösungsverfahren für verschiedene Anwendungen können unterschiedlich gut parallelisiert werden. Eine Leistungssteigerung ist allein durch Vergrößern der Prozessoranzahl nicht zu garantieren, da z.B. gleichzeitig der Aufwand für die notwendige Kommunikation oder die zusätzliche Verwaltung ansteigen kann.

Zur Bewertung massiv-paralleler Algorithmen kann eine Reihe von Kriterien aufgestellt werden. Als die zwei wichtigsten Kriterien sind zu nennen:

- die *Effizienz:* Gefragt sind dabei schnelle Algorithmen (im allgemeinen) und eine gute Ausnutzung der vorhandenen Prozessoren (bei der Parallelität)
- die *Skalierbarkeit*: Sie fordert innerhalb gewisser Schranken eine gleichbleibend hohe Effizienz, auch bei wachsender Anzahl eingesetzter Prozessoren.

Beide Bewertungskriterien werden in der restlichen Arbeit bei der Entwicklung und Untersuchung der parallelen Algorithmen zugrunde gelegt.

1.2 Branch-and-bound

Als grundlegenden Ansatz zur Lösung der zeitkritischen Planungsprobleme werden hier allgemeine Optimierungsverfahren betrachtet. Dazu zählen die Dynamische Programmierung, die Lineare Ganzzahl-Optimierung und das Branch-and-bound-Verfahren. Mit diesen Verfahren können prinzipiell alle geläufigen diskreten Optimierungsprobleme gelöst werden. Die allgemeinen Verfahren sind gegenüber den speziellen Methoden unter Umständen weniger leistungsstark, denn sie können nicht auf eine bestimmte Problemstellung eingehen. Da aber die speziellen Verfahren in der Regel keine allgemeinen Aussagen über die Parallelisierung im Optimierungsbereich direkt zulassen, sollen sie hier nicht untersucht werden.

Bei den allgemeinen Optimierungsansätzen kann man zwischen exakten und approximativen Verfahren unterscheiden. Als approximative Ansätze, die

---

[1] Die Rechenzeit einer *fest vorgegebenen* Problemlösung kann durch parallele Verarbeitung im Grenzfall maximal bis auf den inhärent sequentiellen Anteil reduziert werden (Amdahls law) [Gustafson88].

insbesondere auf Parallelrechnern gut einzusetzen sind, können z.B. Neuronale Netze, Genetische Algorithmen oder Simulated Annealing genannt werden. Leider erzielen sie auch bei kleinen Problemgrößen keine garantiert optimale Lösung. Es ist erstrebenswert, exakte Ansätze einzusetzen und sie gegebenenfalls so zu erweitern, daß sie auch Näherungslösungen berechnen können.

Von den allgemeinen exakten Optimierungsverfahren wird für die Untersuchung der Branch-and-bound-Ansatz (B&B) betrachtet. Das aus dem Operations Research bekannte Verfahren gibt es in vielerlei Varianten. In der Künstlichen Intelligenz findet es sich z.B. als A*-Algorithmus zur Suche kürzester Wege wieder [Nilsson82]. Der B&B bietet eine Reihe von Vorteilen. Neben der Exaktheit des Verfahrens ist auch die Berechnung von suboptimalen Lösungen möglich, deren Kosten innerhalb eines bekannten Abstands zum Optimum liegen (ε-Approximation) [Li84a]. Darauf aufbauend kann ein realzeitfähiger Algorithmus angegeben werden [Wah84b, Korf90].[2] Schließlich zeichnet sich der B&B durch seine leichte Parallelisierbarkeit aus.

Um den Branch-and-bound-Algorithmus anwenden zu können, muß das zu lösende Problem in folgender Form darstellbar sein [Horowitz78]: Gesucht ist ein Vektor $x = (x_1,..., x_n)$, mit ganzzahligen Komponenten $x_i \in \mathbb{N}$, der eine Kriterienfunktion $f(x)$ minimiert (oder maximiert). Zusätzlich ist eine Menge von Randbedingungen gegeben, die von dem Lösungsvektor erfüllt werden müssen. Diese Menge kann unterteilt werden in explizite und implizite Bedingungen. Die expliziten Bedingungen geben die Wertebereiche der einzelnen Komponenten an, die impliziten Bedingungen beschreiben die Abhängigkeiten zwischen den einzelnen Variablen x_i.

Die Menge der Vektoren, welche die expliziten Randbedingungen erfüllen, ergeben einen Suchraum, der als Baum anschaulich dargestellt werden kann (Abb. 1.1). Dabei repräsentieren die inneren Knoten die Vektoren mit nur teilweise instanziierten Komponenten, d.h. noch ungelösten Teilprobleme. Die Kanten des Suchbaums repräsentieren die Bestimmung des konkreten Werts einer Vektorkomponente, d.h. alternative Entscheidungen bezüglich eines offenen Teilproblems. Die betrachteten Suchbäume können durch ihren (mittleren oder maximalen) Verzweigungsfaktor b und durch ihre (maximale) Tiefe d charakterisiert werden.

Bei dem B&B wird das Problem sukzessive in disjunkte Teilprobleme aufgeteilt (branching) und nur noch diejenigen Teilprobleme werden weiter betrachtet, die eine bessere Lösung versprechen als die derzeit beste Lösung (bounding). Zur Zwischenspeicherung der zu betrachtenden Teilprobleme (Knoten des Suchbaums) wird als Datenstruktur die Prioritätsliste "OPEN" eingeführt (Abb. 1.1). Die benötigten Operationen auf dieser Datenstruktur sind das Ein- und Ausfügen von Knoten und die Minimumbildung. Dabei wird ein

---

[2] Dabei wird vorausgesetzt, daß eine erste heuristische Lösung hinreichend schnell gefunden werden kann.

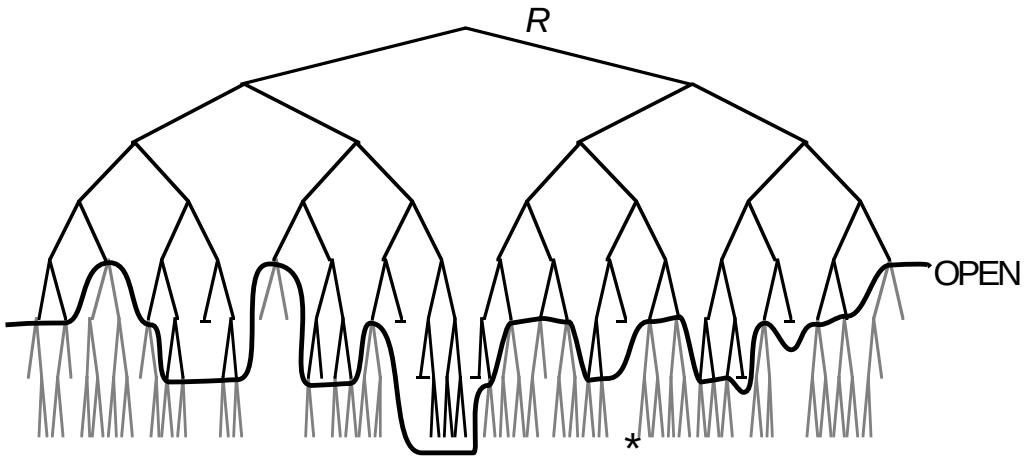

Abb. 1.1: Abarbeitung eines Suchbaums mit Wurzel R *und einer optimalen Lösung (*) durch das Branch-and-bound-Verfahren (B&B) basierend auf der Datenstruktur OPEN*

Knoten mit der minimalen Bewertung gesucht und ausgegeben. Eine effiziente Implementierung von Prioritätslisten stellen sortierte vollständige Binärbäume (Heaps) dar [Horowitz78].

Der Algorithmus 1.1 gibt die Vorgehensweise des B&B als Pseudocode an. Zu Beginn des B&B wird das Ursprungsproblem *R* als Wurzel des Suchbaums in die OPEN-Menge eingefügt. Dies geschieht in der Funktion Init_Branch_and_-bound(). Dort kann auch eine erste heuristische Lösung *z* berechnet werden. In jeder Iteration wird durch die Auswahlfunktion nach einer bestimmten Strategie, z.B. Bestensuche (best-first) oder Tiefensuche (depth-first), ein Knoten *x* aus OPEN ausgewählt und entnommen. Die Expansionsfunktion generiert von *x* alle seine Nachfolger *y*. Anhand einer heuristischen Bewertungsfunktion *g(y)* wird jedem Nachfolger *y* ein Wert zugewiesen. Dabei stellt die Funktion *g(y)* eine untere Schranke für die Kosten *f(y)* der vervollständigten Lösung des Teilproblems *y* dar und ist bzgl. der Nachfolgerrelation der Suchbaumknoten monoton wachsend. Falls die Bewertung *g(y)* günstiger ist als die derzeit beste Lösung *z*, dann wird der Nachfolger *y* in OPEN eingefügt. Falls *y* eine bessere, zulässige Lösung darstellt als die derzeit beste Lösung *z* (incumbent), dann wird diese ersetzt. Andernfalls wird der Nachfolger *y* verworfen. Dieser Mechanismus nennt sich Beschneidung (pruning) des ursprünglichen Suchbaums. Der B&B terminiert, wenn sich keine betrachtenswerte Knoten mehr in OPEN befinden und damit die derzeit beste Lösung *z* als globales Optimum ausgegeben werden kann. (Die verwendeten, problemabhängigen Funktionen sind als Anhang im Kapitel 7.2 angegeben.)

Algorithmus 1.1: (Branch-and-bound)

```
procedure Branch_and_bound(R :node);
/* INPUT:    Wurzel R eines Suchbaums                          */
/* OUTPUT:   optimale Lösung z, falls existent, "failure" sonst */
var set       OPEN;    /* Menge ungelöster Teilprobleme       */
    node      x, y,    /* x, y Knoten des Suchbaums           */
              z;       /* z derzeit beste, zulässige Lösung   */
begin
    if Solution(root) then z := R;
    else z := ø;                    /* mit g(z) = ∞ */
    Init_Branch_and_bound(OPEN, R, z);
    while OPEN contains node x with g(x)<f(z) do begin
        x := Select(OPEN);
        for all successors y of x do begin
            if Solution(y) and f(y)<f(z) then
                z := y;
            if not Leaf(y) and g(y)<f(z) then
                OPEN := OPEN ∪ {y};
        end;
        if Trigger(OPEN) then     /* nur für paralleles B&B */
            Balance(OPEN);
    end;
    if Solution(z) then Return(z);
    else Return(failure);
end;
```

1.3 Feinkörnige Parallelität

Als Verarbeitungsmodell wird die Kategorie Single-Instruction-Multiple-Data (SIMD) nach der Flynnschen Klassifikation zugrunde gelegt [Flynn72]. Dabei wird ein Befehlsstrom auf unterschiedliche Datenströme angewendet. Die Abarbeitung des Algorithmus erfolgt *synchron*, d.h. alle Prozessoren arbeiten denselben Befehlsstrom mit einem gemeinsamen Takt ab (lock step mode). Außerdem ist die Verarbeitung daten-parallel, d.h. die Parallelisierung erfolgt durch gleichzeitige Bearbeitung unterschiedlicher Daten. Die entsprechende Rechnerstruktur ist in Abb. 1.2 dargestellt. Hier wird außerdem von einer *feinkörnigen* Parallelität ausgegangen, d.h. es steht eine sehr große Anzahl (mehrere 10.000) relativ leistungsschwacher Prozessoren zur Verfügung. Die Kommunikation zwischen den Prozessoren wird über ein logisches Verbindungsnetzwerk mit fester Topologie ermöglich. Beispiele für diesen Rechnertypus stellen die Connection Machine CM-2 oder die hier verwendete MasPar MP-1 dar.

Der von den Prozessoren synchron abgearbeitete Befehlsstrom kann in SIMD-Rechnern von einer zentralen Instanz aufbereitet werden. Im Vergleich zu

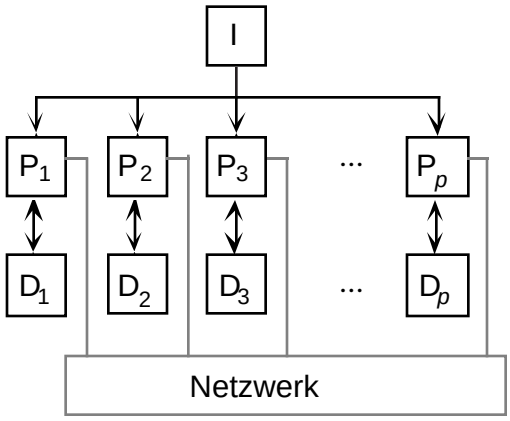

Abb.1.2: Rechnerstruktur für die SIMD-Verarbeitung mit Befehlsstrom I, den Prozessoren P_1 bis P_p, den Datenströmen D_1 bis D_p und einem nicht weiter bestimmten Verbindungsnetzwerk

den Multicomputersystemen (MIMDs) sind die Prozessoren einfacher und damit kleiner. Bearbeitet ein MIMD-Rechner ein Problem, bei dem alle Prozessoren den gleichen Befehlsablauf vollziehen, so erledigen die Steuerwerke der Prozessoren alle redundant die gleiche Arbeit. Die Idee des SIMD-Rechners ist es, die in dieser Redundanz verlorene Chipfläche zu sparen und zugunsten einer höheren Prozessoranzahl zu nutzen. Daher lassen sich SIMD-Rechner mit geringer Menge an Chipfläche und damit auch mit weniger technischem Aufwand bei gleicher Rechenwerkanzahl realisieren. Zu berücksichtigen ist jedoch, daß diese wirtschaftlichen Nachteile durch die Verwendbarkeit von Standardprozessoren geschmälert werden. Abschließend läßt sich erkennen, daß es unmöglich ist, eine global bewertende Entscheidung zugunsten von MIMD oder SIMD zu treffen. Diese Entscheidung muß im Einzelfall für jede Algorithmenklasse und jedes Anwendungsproblem getroffen werden.[3]

Ein wichtiges Kriterium, welches die Effizienz von vielen Algorithmen beeinträchtigt, ist das Zeitverhalten der Kommunikation. Hier ist vor allem das Verhältnis der Aufbauzeit (setup time) zur Übertragungsrate (transfer rate) einer Kommunikationsverbindung zwischen zwei Prozessoren von Interesse. Die *lose-gekoppelten* (loosely coupled) Parallelrechner haben eine lange Aufbauzeit im Vergleich zur Übertragungsrate. Dieses Verhältnis tritt heute zumeist bei dem Austausch von Nachrichten (message passing) als Kommunikations-mechanismus auf, welcher vor allem in MIMD-Rechner zu finden ist. Dagegen haben die *eng-gekoppelten* (tightly coupled) Parallelrechner eine relativ kurze Aufbauzeit der Kommunikationsverbindung. Dies kann die Auswirkung der Kommunikation über einen gemeinsamen Speicher sein, welche heute im allgemeinen in den feinkörnigen SIMD-Rechnern realisiert wird. In dieser Arbeit wird vor allem die Eigenschaft der engen Kopplung bei feinkörnigen SIMD-Rechner ausgenutzt.

---

[3] Eine eigene Parallelrechnerarchitektur für Suchprobleme wurde in [Wah84b] entwickelt.

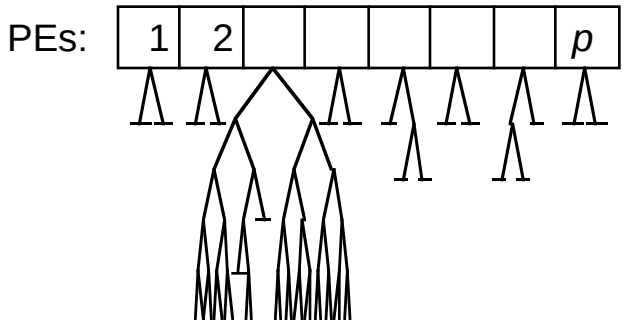

Abb. 1.3: Parallele Abarbeitung des Suchbaums durch Ablauf eines sequentiellen Branch-and-bound-Verfahrens auf jedem der P *Prozessoren*

Zur Parallelisierung des B&B auf Knotenebene können zwei grundverschiedene Vorgehensweisen unterschieden werden: die horizontale und die vertikale Parallelisierung [Roucairol88]. Bei der *horizontalen* Parallelisierung wird die OPEN-Menge, z.B. durch einen Master-Prozessor oder im Shared-Memory, zentral verwaltet. Über einen Scheduler wird die Arbeit an die (Slave-) Prozessoren verteilt.[4] Einerseits kann dadurch eine globale Bestensuche einfach verwirklicht und alle Prozessoren gleichmäßig beschäftigt werden. Die derzeit beste Lösung wird zentral immer auf dem aktuellen Stand gehalten. Andererseits läßt sich zwar bis zu einer bestimmten Prozessoranzahl eine annähernd lineare Beschleunigung (speedup) erreichen, aber darüber hinaus ist keine Verbesserung mehr möglich, da der Zugriff auf die zentrale Datenstruktur zu einem Engpaß wird [Huang90]. Damit wird das Kriterium der Skalierbarkeit schon sehr früh nicht erfüllt.

Bei den hier betrachteten massiv-parallelen Systemen ist daher nur die *vertikale* Parallelisierung sinnvoll (Abb. 1.3). Dabei hat jeder Prozessor eine eigene, unabhängige Datenstruktur OPEN. Nach einer statischen Lastverteilung (Kapitel 3) führt jeder Prozessor den sequentiellen B&B auf seiner eigenen Datenstruktur aus, bis alle Teilprobleme bearbeitet sind. Über Kommunikationskanäle findet der notwendige Informationsaustausch, wie z.B. die derzeit beste Lösung, Terminierungsbedingungen oder die Lastverteilung, statt. Für nachrichten-gekoppelte Multiprozessorsysteme gibt es in der Literatur eine Vielzahl von Vorarbeiten.[5] Für die hier betrachteten feinkörnigen SIMD-Rechner wird auf die allgemeinere Form der parallelen Baumsuche in Kapitel 2.3 noch genauer eingegangen.

Im Gegensatz zum sequentiellen B&B-Algorithmus kann man bei der parallelen Ausführung des B&B drei verschiedene Phasen der Bearbeitung beobachten: die Aufbau-, die Bearbeitungs- und die Beendigungsphase. In der

---

[4] Horizontal-parallelisiertes B&B ist zu finden in: [Abdelrahman88, Imai79b, Janakiram88, Mohan83, Sprague90] .

[5] Vertikal-parallelisierte B&B für MIMDs sind zu finden in: [Arvindam89, ElDessouki80, Huang90, Kumar88, Lai85a, Li84a, Miller89, Pargas88, Quinn86, Taudes91, Vornberger87, Wah84b]

Aufbauphase beginnt der B&B mit der Wurzel den Suchbaum zu expandieren. Dies geschieht solange, wie sich weniger Knoten als Prozessoren in OPEN befinden. Dadurch ist zwangsweise eine Reihe von Prozessoren unbeschäftigt. Sie warten bis sie auf irgendeine Art Knoten zur Bearbeitung erhalten. In der *Bearbeitungsphase* enthält die Datenstruktur OPEN genug Knoten und sehr viele Prozessoren sind mit Arbeit versorgt. In der *Beendigungsphase* sind wieder einige Prozessoren unbeschäftigt, da nur noch wenige Zweige des Suchbaums zur Bearbeitung verblieben sind und die Knotenanzahl kleiner als die Prozessoranzahl ist.

1.4 Lastverteilung

Bei der parallelen Ausführung des B&B arbeiten nach einigen Iterationen nicht mehr alle Prozessoren weiter. Dafür gibt es zwei mögliche Gründe. Einerseits kann die entsprechende lokale Datenstruktur OPEN leer sein, d.h. der Prozessor hat keine Knoten mehr zu bearbeiten. Andererseits kann die Speicherkapazität von OPEN erschöpft sein, d.h. weitere Nachfolgerknoten können nicht mehr aufgenommen werden. In beiden Fällen ist eine Umverteilung der Knoten notwendig, so daß eine größtmögliche Anzahl von Prozessoren weiterarbeiten kann. Nur durch effektives und effizientes Lösen dieses Problems ist eine gute Ausnutzung der vorhandenen Parallelität möglich. Diese Aufgabe wird von der *Lastverteilung* übernommen.

Einen Überblick und eine Klassifikation der Methoden zur Lastverteilung wird in [Casavant88, Schabernack92] gegeben. In [Williams91] werden typische Lastverteilungsmethoden verglichen. Die Ergebnisse zeigen, daß die Durchführung der Lastverteilung einen signifikanten Anteil der Rechenzeit benötigen kann. Bezüglich der Aufgabe der Lastverteilung können zwei Ziele unterschieden werden: Load-sharing und Load-balancing. Das Ziel *Load-sharing* beinhaltet die Wiederbeschäftigung möglichst vieler stillstehender Prozessoren. Bei dem Ziel *Load-balancing* wird zusätzlich eine Gleichverteilung der Last über alle Prozessoren angestrebt.

Ein wesentliches Merkmal zur Unterscheidung der Methoden ist der Zeitpunkt, an dem die Entscheidungen der Lastverteilung getroffen werden. Bei den *statischen* Methoden muß die anfallende Last im voraus bekannt sein und wird vor Beginn des Anwendungsalgorithmus verteilt. Damit ist eine optimale Verteilung potentiell möglich. Bei den *dynamischen* Methoden ist über die anfallende Last im voraus nichts oder nur sehr wenig bekannt und sie muß daher während der Laufzeit der Anwendung verteilt werden. Die Veränderung der Prozessorzustände stellt dabei die Eingabeinformation der Lastverteilung dar.

Ein weiteres Merkmal von Lastverteilungsmethoden ist der Ort der Entscheidungen. Die *zentralen* Methoden entscheiden über die Lastverteilung genau an einer einzigen Stelle (Prozessor) im System. Nachteilig sind vor allem

bei größeren Systemen die durch diesen Engpaß auftretenden Konsequenzen: Es kommt zu Stauungen am Entscheidungszentrum und hohen Kommunikation- und Kalkulationsaufwand. Bei den *verteilten* Methoden wird die Lastverteilung an mehreren Stellen vorgenommen. Für die meisten Anwendungen massiv-paralleler Algorithmen ist dies der einzig mögliche Ansatz.

Im Gegensatz zu einigen anderen Algorithmen ist beim parallelen B&B das Problem der Lastverteilung besonders schwierig. Zum einen ist die aus einem Knoten im B&B entstehende Last nicht vorhersehbar, da der von diesem Knoten repräsentierte Teilbaum noch nicht expandiert wurde und daher seine Größe unbekannt ist. Zum anderen ist das Lastaufkommen während des B&B-Prozesses extrem dynamisch. Dies ist in der heuristischen Beschneidung ungünstiger Knoten begründet. Da jeder Lastverteilungsschritt auf SIMD-Rechner eine Kommunikation aller Prozessoren bedarf, wurde bis vor kurzem die Abarbeitung von Suchalgorithmen auf SIMD-Rechnern als ungeeignet eingestuft [Kindervater89, Frye90]. Diese Einschätzung konnte inzwischen durch andere Arbeiten, welche globale Lastverteilungsmechanismen einsetzen, widerlegt werden (Kapitel 2.3).

Zur genaueren Spezifikation des Lastverteilungsproblems werden hier diskrete Last- bzw. Arbeitseinheiten angenommen. In dem parallelen B&B stellen die einzelnen Knoten des Suchbaums diese Lasteinheiten dar. Als Maß für die Belastung eines Prozessors wird die Anzahl von Knoten in der lokalen OPEN-Menge verwendet.

1.5 Anwendungsdomäne

Als konkrete Anwendungsdomäne zur Untersuchung des feinkörnig parallelisierten B&B werden die im Produktionsbereich häufig vorkommenden Problemstellungen der Maschinenbelegungsplanung betrachtet. Eine Einführung in dieses Gebiet wird in [Baker74a] gegeben. Im Bereich der Belegungsplanung gibt es eine Vielzahl unterschiedlicher Varianten. Durch Kombination aller denkbaren Randbedingungen (dargestellt in [Augustin93]) können über 10^9 unterschiedliche Probleme identifiziert werden. Dabei ist ein großer Teil dieser Probleme NP-hart[6]. So wurden schon von über 4.500 untersuchten elementaren Belegungsplanungsproblemen ca. 80 % als NP-hart klassifiziert [Lawler89]. Viele der komplexeren Problemstellungen lassen sich auf diese elementaren Probleme zurückführen.

Der Einsatz von Parallelität ist gerade bei rechenaufwendigen Problemen wichtig. Diese sind vor allem bei der statischen Belegungsplanung zu finden.

---

[6] In deterministisch polynomialer Zeit lösbare Probleme liegen in der Komplexitätsklasse P. Dagegen sind die Probleme aus der Klasse NP in nicht-deterministisch polynomieller Zeit lösbar. Ein bekanntes offenes Problem ist, ob P = NP gilt. Ein Problem heißt NP-hart, falls sich alle Probleme aus NP darauf reduzieren lassen. (Siehe [Garey79])

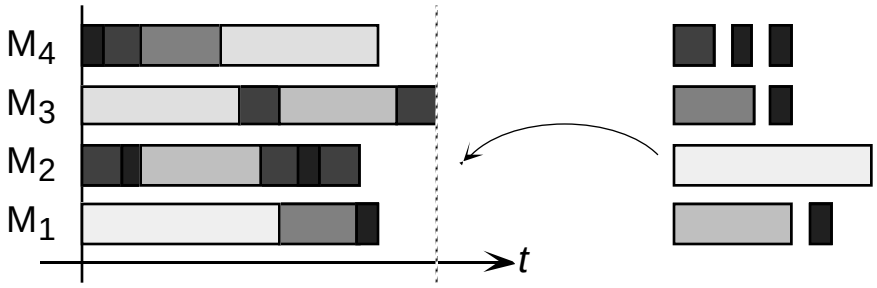

Abb. 1.4: Partieller Maschinenbelegungsplan mit 4 Maschinen (M_1 bis M_4) und 17 schon eingeplanten Jobs als Gantt-Diagramm über der Zeit t *(links) und 8 noch einzuplanende Jobs (rechts)*

Dort ist das Problem von Anfang an im wesentlichen gegeben und spätere Änderungen sind die Ausnahme. Dadurch können kombinatorisch aufwendige Optimierungsprobleme entstehen. Den Gegensatz dazu bildet die dynamische Belegungsplanung. Dort besteht die Aufgabe darin, die einzelnen Aufträge sofort nach ihrer Ankunft einzuplanen. Die Schwierigkeit im Entwurf des Algorithmus liegt in leistungsstarken Heuristiken und nicht in der aufwendigen Optimierung.

Das Grundproblem bei der statischen Belegungsplanung ist die Zuordnung einer Reihe abstrakter Aufträge ("Jobs") auf eine Reihe abstrakter Resourcen ("Maschinen") unter Optimierung vorgegebener Kriterien. Dabei können zwei Aspekte unterschieden werden. Der Aspekt der Reihenfolgeplanung betrifft die zeitliche Abfolge der Jobs auf einer einzelnen Maschine. Der Aspekt der Belegungsplanung betrifft die Zuordnung eines Jobs auf eine bestimmte Maschine.

Das allgemeine Optimierungsverfahren B&B hat in vielen Fällen exponentiellen Aufwand. Dieser Aufwand ist zunächst unabhängig von der Komplexität des bearbeiteten Problems. Andererseits können für Probleme aus der Komplexitätsklasse P spezielle und damit effizientere Verfahren entwickelt werden. Für diese Verfahren wird eine polynomiale oder bessere Laufzeit garantiert. Daher wird hier der B&B anhand eines bekanntermaßen NP-harten Problems untersucht, da dieses mit großer Wahrscheinlichkeit eine exponentielle Komplexität aufzeigt.

Das einfachste Belegungsplanugsproblem, welches NP-hart ist und die beiden oben genannten Aspekte beinhaltet, ist die non-operationale Belegungsplanung von parallelen Maschinen. Mehrere Jobs werden auf parallel angeordneten, gleichartigen Maschinen eingeplant. Die Aufträge sind non-operational (elementar) und können nicht unterbrochen werden. Zwischen den Aufträgen werden keine Reihenfolgebedingungen zugelassen, da sie die Problemkomplexität stark verändern können. Das zu minimierende Kriterium *f(x)* eines (partiellen) Planes *x* ist die Gesamtbearbeitungszeit (makespan) aller Aufträge. Nach der im Operations Research oft verwendeten Klassifikation aus [Lawler89] läßt sich das Problem beschreiben als $P//C_{max}$, wobei *P* für "parallele Maschinen" und C_{max} für die maximale Bearbeitungszeit als Kriterienfunktion

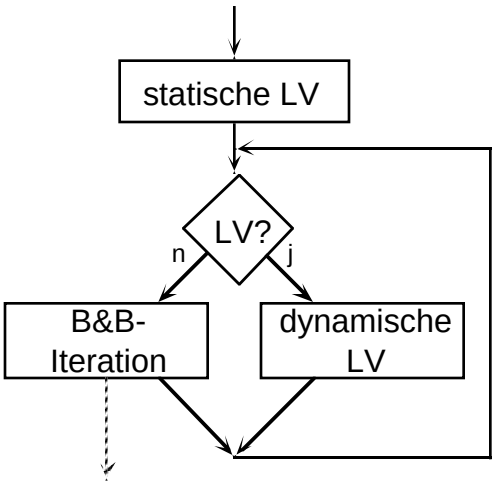

Abb. 1.5: Schematischer Ablauf des parallelisierten Branch-and-bound-Algorithmus (B&B) mit statischer und dynamischer Lastverteilung (LV) und dem Trigger-Mechanismus (LV?)

steht. Dieses Problem ist schon ab zwei Maschinen NP-hart. In [Cheng90] ist ein Literaturüberblick mit bekannten Problemkomplexitäten und speziellen Lösungsmethoden gegeben.

Ein Beispiel für diese Anwendungsdomäne ist in Abb. 1.4 gegeben. Dort ist ein partieller Maschinenbelegungsplan als Gantt-Diagramm dargestellt. Dabei wurde auf den vier Maschinen nur der erste Teil der insgesamt 25 Jobs eingeplant. In dem vom B&B abgearbeiteten Suchbaum entspricht dieser partielle Plan einem inneren Knoten. Die derzeitigen Kosten des Plans, in diesem Fall gemessen in der maximalen Bearbeitungszeit, sind durch die gestrichelte Linie dargestellt. Das Einplanen eines weiteren Jobs auf eine bestimmte Maschine (angedeutet durch den Pfeil) führt entlang einer Kante im Suchbaum zu dem entsprechenden Nachfolgerknoten.

1.6 Zielsetzung und Übersicht

Zusammenfassend wird in dieser Arbeit die Parallelisierung des allgemeinen und exakten Lösungsverfahrens Branch-and-bound (B&B) auf SIMD-Rechnerstrukturen betrachtet. Die in dieser Arbeit angegangene Problemstellung ist die sowohl effiziente als auch skalierbare Lastverteilung für feinkörnig parallelisiertes B&B. Als exemplarische, NP-harte Anwendungsdomäne werden statische, non-operationale Belegungsplanungsprobleme ohne Reihenfolgebedingungen eingesetzt.

Die in der Literatur übliche Vorgehensweise zur Integration der Lastverteilung mit parallelen Baumsuch-Algorithmen kann als Blockdiagramm schematisiert werden (Abb. 1.5). Nach einer mehr- oder weniger stark ausgeprägten statischen Lastverteilung zu Beginn des B&B-Ablaufs werden auf jedem Prozessor der B&B-Algorithmus und die dynamische Lastverteilung als

zwei nebenläufige Prozesse aufgefaßt. Dabei entscheidet ein Trigger-Mechanismus z.B. aufgrund der aktuellen Prozessorauslastung, welcher der beiden Prozesse den nächsten elementaren Schritt durchführen kann. Dies geschieht solange, bis der B&B-Prozeß die Terminierung des B&B erkennt. Dieses Ablaufschema ist sowohl bei MIMD-Rechner mit asynchronen Kommunikationsprotokollen als auch bei SIMD-Rechner mit Lock-step-mode zutreffend.

Anhand dieses Ablaufschemas wird nun ein Überblick über die in dieser Arbeit verfolgten Ansätze zu der oben genannten Problemstellung gegeben. Nach einem Abriß über den Stand der Forschung in Kapitel 2 geht die Arbeit auf folgende drei Ansatzpunkte zur Lastverteilung für paralleles B&B ein:[7]

Die *statische Lastverteilung* in Kapitel 3 wird vor dem eigentlichen B&B-Algorithmus ausgeführt. Sie strebt eine Versorgung aller Prozessoren mit Arbeit durch ein anfängliches Expandieren und Abarbeiten des Suchbaums an. Es wird erstmanls eine Reihe von Verfahren herausgestellt und auf ihre Eignung hin verglichen.

Die *dynamische Lastverteilung* in Kapitel 4 arbeitet nebenläufig zu dem Anwendungsalgorithmus und versucht die dort entstehenden Unausgewogenheiten in der Verteilung der Arbeit wieder auszugleichen. Dazu wird ein vereinfachtes, gut skalierbares Flüssigkeitsmodell als erste synchrone lokale Lastverteilung entwickelt. Die Methode wird mit dem bekannten, aus dem Asynchronen übertragenen Mittelungs-Ansatz verglichen. In diesem Kapitel wird auch der notwendige Trigger-Mechanismus besprochen.

Die *implizite Lastverteilung* in Kapitel 5 ist eine Verschmelzung des B&B mit dem Lastverteilungsprozeß, so daß durch die veränderte Bearbeitung des Suchbaums die Last automatisch auf die Prozessoren verteilt wird. Es werden dabei die vier Blöcke aus Abb. 1.5 zu einem Block integriert. Das neuartige Konzept der k-Expansion unterstützt eine automatische Lastverteilung und approximiert eine globale Suchstrategie.

Nach diesen drei Hauptkapiteln werden in Kapitel 6 die in dieser Arbeit erzielten Ergebnisse zusammengefaßt und diskutiert. Zur Vollständigkeit geben die Anhänge in Kapitel 7 eine Reihe von Detailinformationen an. Dabei handelt es sich um die Liste der verwendeten Benchmark-Probleme, um die Beschreibung der problemabhängigen Funktionen des B&B und um die Zeitmessung von u.a. diesen Grundoperation. Schließlich wird noch auf die Register, insbesondere auf das Schlagwortverzeichnis (inklusive den verwendeten Abkürzungen), am Ende der Arbeit in Kapitel 8 verwiesen.

---

7 Die Piktogramme geben die betroffenen Blöcke des Ablaufschema in Abb. 1.5 an.

2. Stand der Forschung

In diesem Kapitel wird eine Reihe von synchronen Baumsuchalgorithmen sowie dynamischen Lastverteilungsmethoden aus der Literatur vorgestellt. Dagegen wird auf die statische Lastverteilung für Baumsuche nicht eingegangen. Dort wird der Suchbaum erst sukzessive aufgebaut und abgearbeitet, und der Umfang der zukünftigen Arbeit ist nicht bekannt. Daher ist bei dem parallelen B&B keine statische Lastverteilung im herkömmlichen Sinn möglich. Aus diesem Grund kann man sich bei dem Literaturüberblick auf rein dynamische Lastverteilungen beschränken. Was dennoch durch anfängliches Abarbeiten des Suchbaums an statischer Lastverteilung beim parallelen B&B möglich ist, wird in Kapitel 3 untersucht.

In Kapitel 2.1 werden allgemeine Ansätze zur Lastverteilung auf SIMD-Rechnern beschrieben. Obwohl diese Ansätze teilweise für die Baumsuche entwickelt wurden, sollen sie hier als eigenständige Lastverteilungen dargestellt werden. In Kapitel 2.2 werden dann lokale Verfahren zur Lastverteilung besprochen. Dabei kommen sowohl synchrone als auch asynchrone Verfahren vor. Schließlich wird in Kapitel 2.3 auf die Baumsuche mit SIMD-Verarbeitung eingegangen. Für die schon im ersten Unterkapitel vorgestellten Verfahren werden nur die für die Baumsuche typischen Merkmale herausgestellt.

Zunächst soll über die besprochenen Lastverteilungsmethoden ein Überblick gegeben werden. Dazu wird einerseits zwischen Methoden für Rechner mit lose- und eng-gekoppelten Prozessoren unterschieden. Diese Eigenschaften korrespondieren in der Praxis zumeist mit der MIMD- und SIMD-Verarbeitung. Andererseits wird zwischen Lastverteilungsmethoden mit globaler, semi-lokaler und lokaler Kommunikation differenziert. Die nach diesen Eigenschaften resultierende Klassifikation der Lastverteilungsmethoden ist in Tabelle 2.1 gegeben. Eine detailierte Beschreibung der Methoden erfolgt dann in den einzelnen Unterkapiteln.

	Lose-gekoppelt (MIMD-Rechner):	Eng-gekoppelt (SIMD-Rechner):
Globale LV:	…	Rendezvous-allocation (2.1.2) Trigger: S^x, D^P, D^K (2.1.1)
Semi-lokale LV:	Gradienten-Modell (2.3.4) Hierarchische Diffusion (2.3.4)	Scan-directed-load-balancing (2.1.2)
Lokale LV:	Diffusionsansatz (2.2.1) Dimensions-Austausch (2.2.2) Nachbarschafts-Mittelung (2.2.3) Partikelansatz (2.2.4)	Flüssigkeitsmodell (4)

Tabelle 2.1: Überblick über die dynamischen Lastverteilungsmethoden gegliedert nach Lokalität der Verfahren und Kopplung der Prozessoren (Kapitelangaben in Klammer und bekannte Verfahren grau unterlegt)

2.1 Lastverteilung für SIMD-Rechner

Für MIMD-Rechner wurden schon viele Ansätze zur Lastverteilung untersucht (siehe z.B. [Kumar91]). Aber erst in den letzten Jahren wurden parallele Suchverfahren auch für SIMD-Rechner entworfen. Dabei muß auch eine synchrone Lastverteilung betrachtet werden. Im folgenden werden zunächst Mechanismen zur Initiierung der Lastverteilung vorgestellt. Danach folgen zwei Verfahren zur synchronen Lastverteilung.

2.1.1 Trigger-Mechanismen

Bevor die eigentliche Lastverteilung durchgeführt wird, muß bestimmt werden, welcher Zeitpunkt dafür am günstigsten ist. Gerade auf SIMD-Architekturen, bei denen alle Prozessoren gleichzeitig an der Lastverteilung beteiligt sind, ist der Zeitpunkt der Lastverteilung von großer Bedeutung.

Ein einfacher und intuitiver Mechanismus aus [Powley93, Frye90] löst die Lastverteilungsphase aus, wenn die Anzahl aktiver Prozessoren A kleiner als ein festen Bruchteil $x \in [0,1]$ der Gesamtzahl P aller Prozessoren wird. Dieser sogenannte *statische* Trigger S^x initiiert die nächste Lastverteilungsphase, falls gilt

$$A \leq x \cdot P \tag{2.1}$$

Für den statischen Trigger S^x stellt sich die Frage, welcher Schwellwert x die maximale Effizienz des Gesamtalgorithmus erbringt. Basierend auf der

Gesamtarbeit wird in [Karypis92] analytisch der optimale Schwellwert x_{opt} berechnet. Leider kann man aber die Gesamtarbeit bei Suchproblemen nur selten genau abschätzen.

Alternativ zu dem statischen Trigger kann auch ein *dynamischer* Trigger eingesetzt werden. Dabei wird eine Schwelle verwendet, deren Wert sich an die Charakteristik des Problems über die Zeit anpaßt. Der Trigger versucht den Gewinn gegen Verlust bei der Durchführung einer Lastverteilung abzuwägen. Dabei stellt der Gewinn die Zeitreduktion durch die Wiederbeschäftigung der stillstehenden Prozessoren dar. Der Verlust entspricht der für die Lastverteilung benötigten Zeit, da dort bei synchroner Verarbeitung keine produktive Arbeit geleistet werden kann.

Ein solcher dynamischer Trigger D^P wurde in [Powley93] vorgeschlagen und analysiert. Sei dazu T_{active} die Summe der Arbeitszeit aller Prozessoren und $T_{B\&B}$ die vergangene Zeit nach der letzten Lastverteilungsphase, und sei T_{LV} die erforderliche Zeit um in der nächsten Lastverteilung alle Prozessoren wieder zu beschäftigen. Die nächste Lastverteilung wird nach dem Trigger D^P initiiert, falls durch die Lastverteilung der Gewinn $T_{active} - A \cdot T_{B\&B}$ größer als der Verlust $A \cdot T_{LV}$ ist, also falls gilt

$$T_{active} \geq A \cdot (T_{LV} + T_{B\&B}) \tag{2.2}$$

Ein anderer dynamischer Trigger D^K wird in [Karypis92] vorgestellt. Dieser erbringt eine Effizienz im Bereich des optimalen, statischen Triggers S^x. Dabei wird die Summe der Stillstandszeit aller Prozessoren T_{idle} seit der letzten Lastverteilung mit den Kosten $T_{LV} \cdot P$ der nächsten Lastverteilung ausgeglichen. Damit initiiert der Trigger D^K eine Lastverteilung, falls gilt

$$T_{idle} \geq T_{LV} \cdot P \tag{2.3}$$

Für beide dynamischen Trigger D^P und D^K ist die benötigte Zeit T_{LV} für die nächste Lastverteilungsphase nicht bekannt. Sie wird daher durch den Wert aus der letzten Phase approximiert.

Alle drei oben beschriebene Trigger-Mechanismen haben einen gemeinsamen Nachteil. Für ihre Anwendung wird globale Information über den Lastzustand aller Prozessoren benötigt. Bei geringen Prozessoranzahlen ist dies noch hinreichend schnell zu bewerkstelligen. Aber für eine beliebig wachsende Anzahl von Prozessoren skalieren diese Mechanismen nicht mit. Daher wird in Kapitel 4 ein strikt lokaler Trigger-Mechanismus untersucht und für die Experimenten verwendet.

2.1.2 Rendezvous-allocation-method

Eine typische globale Lastverteilung baut auf einem leistungsstarken Mechanismus zur Vermittlung von Nachrichten (router) und der *Rendezvous-*

allocation-method aus [Christman83][1] auf. Dabei werden die wartenden Prozessoren auf die beschäftigten Prozessoren mit einer 1-zu-1-Zuordnung abgebildet. Dies geschieht durch lineares Aufzählen der Prozessoren separat in beiden Gruppen. Dann wird jeder beschäftigte Prozessor dem wartenden Prozessor mit dem gleichen Aufzählungsindex zugeordnet. Die beschäftigten Prozessoren teilen ihre Arbeit in zwei Teile auf und transferieren mit der Rendezvous-allocation-method gleichzeitig einen Teil an ihren korrespondierenden wartenden Prozessor. Falls die Anzahl der wartenden Prozessoren B größer als die der überbelasteten A ist, können nur die ersten A wartenden Prozessoren auf die beschäftigten abgebildet werden, und die verbleibenden $B - A$ Prozessoren erhalten keine Arbeit.

Diese zumeist verwendete Variante hat den Nachteil, daß bei $A > P/2$ immer dieselben ersten Prozessoren Arbeit abgeben und somit bald selbst unbeschäftigt sind. Dagegen wird in [Karypis92] ein globaler Zeiger eingeführt, der angibt, welcher beschäftigte Prozessor zuletzt[2] Arbeit abgegeben hat. Bei jeder Lastverteilung, wird nun die Zuordnung mit dem nächsten beschäftigten Prozessor hinter dem Zeiger begonnen. Falls der Zeiger den letzten Prozessor erreicht, wird wieder mit dem ersten fortgefahren. In [Karypis92] wird analytisch und experimentell gezeigt, daß diese Lastverteilung keine schlechtere Skalierbarkeit aufweist als die besten bekannten Lastverteilungen auf MIMD-Architekturen.[3]

Die Rendezvous-allocation-method als Lastverteilung wurde für einige Algorithmen angewendet. Neben den Baumsuchverfahren (Kapitel 2.3) wird sie in [Gerogiannis93] bei der Merkmalsextraktion aus Bildern in einer erweiterten Form auf den Rechnern iPSC/2 und CM-2 eingesetzt. In der ersten Phase werden Lastpakete der Größe $\overline{L}$ verschickt. Hierbei gibt $\overline{L}$ die mittlere Belastung aller Prozessoren an. Auf dem asynchronen iPSC/2 sorgt dann eine zweite Phase für den feineren Lastausgleich.

2.1.3 Scan-directed-load-balancing

Ein weiterer Lastverteilungsmechanismus stellt das *Scan-directed-load-balancing* (SDLB) von [Biagioni91] dar. Es wird dort für die kanten-basierte Diffusion in der Bildverarbeitung auf einer MasPar MP-1 eingesetzt. Der Mechanismus ist lokalität-erhaltend, d.h. logisch benachbarte Lastelemente sind nach der Lastverteilung wiederum benachbart oder liegen auf demselben Prozessor. Außerdem ist diese Lastverteilung besonders für regelmäßige Berechnungen in Gitter-Topologien geeignet. Das SDLB baut auf der effizienten Ausführung von parallelen Präfix-Operationen (scans) auf, wie sie vor allem

---

1 Die Beschreibung der Rendezvouz-allocation-method und die Referenz stammen aus [Hillis85].

2 "Zuletzt" oder "hinter" bezeichnen hier keine zeitliche oder geometrische Relation, sondern beziehen sich auf die Indizierung der Prozessoren.

3 Je nach Netzwerktopologie: Global-round-robin [Kumar91] bzw. Random-polling [Sanders94],

durch SIMD-Rechner unterstützt werden. Mit Hilfe der Präfix-Operationen wird innerhalb einer Dimension eine Art globaler "Fluß" für die Prozessorbelastung berechnet und damit die korrekte Transferrichtung der Lasteinheiten gesteuert.

Für ein eindimensionales Gitter mit P Prozessoren läßt sich das Verfahren wie folgt formalisieren. Sei der Prozessor i mit einer Last L_i belastet und $\sigma_i = \sum_{j=1}^{i} L_j$ die Präfix-Summe bis Prozessor i. Durch SDLB werden dann in mehreren Teilschritten diskrete Lasteinheiten transferiert, bis ein Fluß von δ_i zwischen dem Prozessor i und seinem linken Nachbarn erfüllt ist. Bei positivem Fluß werden Lasteinheiten von Prozessor i zu Prozessor $i-1$, und bei negativem Fluß in umgekehrter Richtung transferiert. Der Fluß berechnet sich bei einer mittleren Belastung von $\bar{L} = \sigma_P / P$ aus

$$\delta_i = i \cdot \bar{L} - \sigma_{i-1} \tag{2.4}$$

Wie auch bei der Rendezvous-allocation-method wird in SDLB zu einem Teil global gearbeitet. Statt die Lasteinheiten global zu transferieren, wird in SDLB ein globale Information über die Verteilung der Last benötigt. Der Algorithmus hängt damit stark von der Effizienz der Scans und des Broadcasts ab. Außerdem ist die Aufrechterhaltung von Nachbarschaftsbeziehungen der Lastelemente nicht für alle Anwendungen notwendig. Experimente in [Hemminger94] zeigen, daß SDLB auch für den B&B in diesem Sinn mehr als notwendig leistet und damit sehr zeitaufwendig ist.

2.2 Lokale Lastverteilungsmethoden

Neben den Lastverteilungsmethoden für SIMD-Rechner aus dem vorigen Unterkapitel sind für diese Arbeit auch die Methoden für MIMD-Rechner interessant, wobei man sich auf die lokalen Lastverteilungen beschränken kann. Bei *lokalen* Lastverteilungsmethoden wird nur zwischen Prozessoren und ihren direkten Nachbarn kommuniziert. Diese Interpretation unterscheidet sich von der aus [Casavant88], bei der unter lokaler Lastverteilung das Scheduling und Dispatching von Prozessen innerhalb eines einzelnen Prozessors verstanden wird. Andererseits ist der dort verwendete Begriff "verteilte Lastverteilung" nicht präzise genug, denn dabei ist auch die Kommunikation über mehrere unbeteiligte Prozessoren zulässig.

Bei den bekannten lokalen Lastverteilungsverfahren handelt es sich um asynchrone Ansätze, welche allerdings bei der Analyse zumeist von einer synchronen Verarbeitung ausgehen. Im folgenden werden drei streng lokale Lastverteilungsmethoden mit den dazu bekannten Ergebnissen vorgestellt. Es handelt sich um den Diffusionsansatz, den Dimensions-Austausch und die Nachbarschafts-Mittelung. Alle diese Ansätze sind durch die Lokalität in ihrem Vorgehen ähnlich, haben jedoch einen unterschiedlichen Ausgangspunkt. Abschließend wird kurz auf semi-lokale Ansätze eingegangen.

2.2.1 Diffusionsansatz

Der Diffusionsansatz (*diffusion method*) ist ein iterativer Algorithmus und insbesondere für Systeme mit direkten Kommunikationsnetzwerken geeignet. Bei jedem Schritt wird ein fester Anteil der Lastdifferenz zweier direkt benachbarten Prozessoren ausgetauscht. Durch diese lokalen Operationen konvergiert die Verteilung der Last gegen das globale Optimum (gleichmäßige Verteilung). Die Effizienz des Diffusionsansatzes hängt von einem *Diffusionsparameter* α ab, der die Größe des zu transferierenden Anteils angibt.

Für den Prozessor i bezeichne $L_i \in \mathbb{R}$ die Last und $\Delta(i)$ die Menge der direkt benachbarten Prozessoren. Es wird angenommen, daß die Last aus sehr vielen und sehr kleinen Lasteinheiten besteht, so daß eine kontinuierliche Darstellung zulässig ist. Von dem Prozessor i zu jedem Nachbarn $j \in \Delta(i)$ wird nach dem Diffusionsansatz eine Last $\delta_i(j)$ transferiert mit

$$\delta_i(j) = \alpha\,(L_i - L_j), \quad \text{bei } \alpha \in (0,1) \tag{2.5}$$

Für $\delta_i(j) < 0$ findet der Lasttransfer in umgekehrter Richtung statt. Rundungsfehler bleiben dabei unberücksichtigt. Jede Zustandsänderung der Prozessorbelastungen durch die synchronen Lastausgleiche läßt sich mit folgender Übergangsgleichung beschreiben:

$$L_i(t+1) := L_i(t) + \sum_{j \in \Delta(i)} \delta_i(j) \tag{2.6}$$

Für ein System mit P Prozessoren und einer Gesamtbelastung von L, die ungleichmäßig über das System verstreut ist, soll durch den Diffusionsansatz die Belastung der Prozessoren gegen L/P konvergierten. In [Cybenko89] findet die erste Analyse dieses Verfahrens statt. Unter der Annahme von synchroner Kommunikation werden dort hinreichende und notwendige Bedingungen für den Diffusionsparameter zur Konvergenz angegeben. Es wird auch der optimale Parameter für Hyperkuben ermittelt, welcher die schnellste Konvergenzrate für die Lastverteilung ermöglicht. Außerdem wird in [Bertsekas89] gezeigt, daß die asynchrone Version des Diffusionsansatzes konvergiert, vorausgesetzt, daß die Verzögerungszeit durch die Kommunikation auf einer Verbindung nach oben beschränkt ist.

Neben der notwendigen Konvergenz an sich ist auch die Geschwindigkeit der Konvergenz wichtig. Sie gibt an, in welcher Zeit die Lastkonfiguration zu einem ausgeglichenen Zustand konvergiert. In [Boillat90] werden verschiedene Konvergenzraten für unterschiedliche Netzwerk-Topologien angegeben. Dabei wird nur die Anzahl von Kommunikationsschritten, nicht aber die Kommunikationsmenge berücksichtigt. Es wird gezeigt, daß für einen D-dimensionalen Torus mit K_i Prozessoren in der Dimension i die Konvergenzrate asymptotisch $O(D \cdot \max\{K_i\}^2)$ beträgt.[4] Für einen D-dimensionalen, binären

[4] Die Notation des O-Kalküls ist am Ende der Arbeit in der Symboltabelle gegeben (Kapitel 8.2).

Hyperkubus dagegen nur O(D). Außerdem hängt die Anzahl benötigter Iterationen zum Erhalt einer ausgeglichenen Lastkonfiguration, von der initialen Lastkonfiguration ab, welche aber bei dem verwendeten Aufwandsmaß nicht zum Tragen kommt. In [Xu93] wird der optimale Diffusionsparameter für eine synchrone Verarbeitung in (symmetrischen) D-dimensionalen Tori mit K Prozessoren pro Dimension[5] angegeben.

In [Kumar87] wird eine Art empfänger-initiierter Diffusionsansatz unter dem Namen α-Splitting für unidirektionale Ringe analysiert. Falls ein unbelasteter Prozessor mit Index $i + 1$ Last von einem seinem Vorgänger i anfordert, dann wird $(1 - \alpha) \cdot L_i$ Last, mit $0 < \alpha < 1$, von der aktuellen Last L_i an den Prozessor $i + 1$ transferiert. Die Analyse ergibt, daß asymptotisch ein exponentieller Zeitaufwand β^P, mit $\beta = 1/(1 - \alpha)$, bei wachsender Prozessoranzahl P benötigt wird. Der Aufwand wird dabei in der Anzahl von Transfers unabhängig von der (kontinuierlichen) Informationsmenge gemessen. Die Diskrepanz zu den Ergebnissen aus dem letzten Abschnitt erklärt sich durch die unterschiedliche Initiierung der Lastverteilung.

Für den Diffusionsansatz gibt es eine Vielzahl von Anwendungsgebieten. Mit asynchronen B&B-Algorithmen wurde der Diffusionsansatz auf dem Rechner iPSC/2 in [Willebeek90] und auf einem Transputer-Netzwerk mit de-Bruijn- und Ring-Topologien in [Lüling92] eingesetzt.

Ein wesentlicher Nachteil des Diffusionsansatzes zeigt sich in der praktischen Anwendung. In [Horton93] wird aufgezeigt, daß trotz nachgewiesener Konvergenz keine globale Balance im Fall einer lokalen Balance auftreten muß. Dieser Effekt ist in der diskreten Realisierung eines kontinuierlichen Modells begründet. Durch die notwendigen Rundungsfehler können abfallende Treppenkonstellationen mit einem Steigungsverhältnis von –1 auftreten, welche nicht weiter balanciert werden.

2.2.2 Dimensions-Austausch

Eine weitere lokale Lastverteilung ist der Dimensions-Austausch (*dimension exchange*). Dies ist ein synchroner Ansatz, bei dem die Lastverteilung jeweils in einer Dimension des Verbindungsnetzwerks stattfindet (siehe auch [Willebeek93]). In [Cybenko89] wird der Dimensions-Austausch für asynchrone Multiprozessorsysteme mit einer binären Hyperkubus-Topologie präsentiert. Um den balancierten Zustand zu erreichen sind log(P) Lasttransfers bei P Prozessoren nötig. Ein Lasttransfer beinhaltet die Kommunikation von mehreren Lastelementen. Dieser Ansatz ist für Probleme mit kleinen oder fehlenden Abhängigkeiten zwischen den Lastelementen geeignet.

Vergleicht man den Dimensions-Austausch mit dem Diffusionsansatz, dann arbeiten beide Ansätze für verschiedene Kommunikationsmodelle unterschiedlich gut. Für den Diffusionsansatz ist eine simultane

---

5 Dies entspricht einem K-fachen D-Kubus.

Kommunikation mit allen direkten Nachbarn wünschenswert. Bei dem Dimensions-Austausch ist eine Kommunikation zu einem Zeitpunkt ausreichend. In [Cybenko89] wird gezeigt, daß der Dimensions-Austausch für Hyperkuben schneller als der Diffusionsansatz ist. Dies trifft auch für den (symmetrischen) D-dimensionalen Torus mit K Prozessoren in jeder Dimension zu [Xu93]. Dennoch ist der Diffusionsansatz in der Praxis zu bevorzugen, da keine Kantenfärbung notwendig ist, wie bei dem Dimensions-Austausch.

2.2.3 Nachbarschafts-Mittelung

Die Nachbarschafts-Mittelung (*nearest neighbour averaging*, NNA) ist eine weitere, strikt lokale Lastverteilung. Die Idee dabei ist, die Belastung eines jeden Prozessors so zu verändern, daß sie der mittleren Belastung vor der Umverteilung entspricht. Betrachtet man den Prozessor i und die Menge seiner direkten Nachbarn $\Delta(i)$, so ergibt sich zum Zeitpunkt t eine lokale, mittlere Belastung von $\overline{L}_i(t) = (L_i(t) + \sum_{j \in \Delta(i)} L_j(t)) \; / \; (|\Delta(i)| + 1)$. Nach einem Lastverteilungsschritt zum Zeitpunkt $t + 1$ soll durch die Nachbarschafts-Mittelung der Prozessor i mit genau der mittleren Last $\overline{L}_i(t)$ belastet sein. Als Übergangsgleichung für die Belastungsänderung gilt also

$$L_i(t+1) := \frac{1}{|\Delta(i)| + 1} \left(L_i(t) + \sum_{j \in \Delta(i)} L_j(t) \right) \tag{2.7}$$

Diese Nachbarschafts-Mittelung kann synchron oder asynchron erfolgen. Die asynchrone Variante wird in [Willebeek93] beschrieben. Falls ein Prozessor überdurchschnittlich belastet ist, dann transferiert er einen Teil seiner Last an diejenigen Nachbarn mit unterdurchschnittlicher Belastung. Die Menge der übertragenen Last ist proportional zu der Differenz zwischen der mittleren Belastung und der Last des Nachbarn. Sei dazu die Unterbelastung eines einzelnen benachbarten Prozessors $j \in \Delta(i)$ durch $h_j = \max\{0, \overline{L}_i - L_j\}$, und die gesamte Unterbelastung der Nachbarn durch $H_i = \sum_{j \in \Delta(i)} h_j$ ausgedrückt. Dann ergibt sich von Prozessor i zu jedem seiner Nachbarn $j \in \Delta(i)$ nach der asynchronen Nachbarschafts-Mittelung ein Lasttransfer $\delta_i(j)$ mit

$$\delta_i(j) = (\overline{L}_i - L_i) \frac{h_j}{H_i} \tag{2.8}$$

Bei der synchronen Variante der Nachbarschafts-Mittelung wird die Arbeit eines Prozessors in $|\Delta(i)| + 1$ gleich große Stücke aufgeteilt. Der Prozessor selbst und jeder seiner Nachbarn erhält einen Teil der Arbeit. Durch Ausführung dieses Lastverteilungsschritts ist die Übergangsgleichung (2.7) erfüllt. Der Prozessor i sei mit der Last L_i belastet und habe $\Delta(i)$ Nachbarn. Dann findet durch

die synchrone Nachbarschafts-Mittelung von Prozessor i an jeden seiner Nachbarn $j \in \Delta(i)$ ein Lasttransfer δ_i statt mit[6]

$$\delta_i = \frac{L_i}{|\Delta(i)| + 1} \tag{2.9}$$

Die synchrone Nachbarschafts-Mittelung wird in [Hong88] für binäre und in [Qian91] für allgemeine Hyperkuben analysiert. Dabei wird jeweils die Konvergenz der Lastverteilung gezeigt, und daß die Varianz der Belastungen nach oben beschränkt ist.

Im Vergleich mit dem Diffusionsansatz aus Kapitel 2.2.2 stellt sich die Nachbarschafts-Mittelung als ein Spezialfall dar. Für einen Diffusionsparameter $\alpha = 1/(|\Delta(i)| + 1)$ geht die Übergangsgleichung des Diffusionsansatz (2.6) in die der Nachbarschafts-Mittelung (2.7) über.

2.2.4 Der Partikelansatz

In [Heiss93] wird das Problem der dynamischen Zuordnung von Betriebssystems-Tasks auf Prozessoren in lose-gekoppelten MIMD-Systemen angegangen. Neben der Lastverteilung ist das Ziel intensiv kommunizierende Tasks nahe beieinander zu plazieren, um Verzögerungen durch die Kommunikation zu minimieren. Da die Plazierung dynamisch durch Migration der Tasks erfolgt, werden auch die Migrationskosten berücksichtigt. Bei dem lokalen Ansatz werden die Tasks als "Partikel" betrachtet, welche von Kräften beeinflußt werden. Zu den Kräften gehören neben der Lastdifferenz die Kosten für Interprozeßkommunikation und Task-Migration. Jedes Starten und Beenden von Tasks initiiert die Lastverteilung. Die Migrationen stagnieren, wenn eine stabile Situation mit balancierten Kräften erreicht ist.

Der Ansatz wurde für den 8x8-Torus und den 6D-Hyperkubus simuliert. Die Ergebnisse zeigen die Notwendigkeit bei der Lastverteilung den Mehraufwand durch Kommunikation und Migration mitzuberücksichtigen. Aber bei Anwendung dieses Ansatzes auf die hier betrachtete Baumsuche entfällt dieser Mehraufwand. Einerseits kommunizieren die Lastelemente (Suchbaumknoten) nicht miteinander, so daß die Verzögerung durch ungünstige Plazierung nicht auftritt. Andererseits sind alle Knoten des Suchbaums gleich groß, so daß die Migrationskosten konstant sind und entfallen können. Damit reduziert sich der Ansatz für die Baumsuche auf die schon besprochenen Diffusionsansätze (Kapitel 2.2.1).

---

6 Der Kommunikationsaufwand kann durch vorherigen Austausch der Belastungsinformation auf die Differenz zweier entgegengesetzter Lasttransfers reduziert werden.

2.2.5 Semi-lokale Lastverteilung

Neben den oben beschriebenen, strikt lokalen Lastverteilungsmethoden gibt es noch weitere Ansätze. Sie haben gemeinsam, daß die Anforderung der strengen Lokalität mehr oder weniger aufgegeben wird. Dennoch kann man sie nicht ganz zu den globalen Lastverteilungen zählen. Daher sollen sie nur kurz aufgeführt werden.

Ein hierarchischer, topologie-unabhängiger Diffusionsansatz für Multiprozessoren wird in [Horton93] beschrieben. Basierend auf globaler Information über die Lastverteilung werden lokale Lasttransfers gesteuert. Einerseits können geforderte Nachbarschaftsbeziehungen zwischen den Daten berücksichtigt werden. Andererseits müssen auf jedem Prozessor immer genug Lastelemente für die Transfers zu Verfügung stehen. Zur Balance sind $O(\log P)$ Transferschritte bei P Prozessoren notwendig. Hierbei enthält ein Transfer die Kommunikation von mehreren Lastelementen.

Das Gradienten-Modell aus [Lin87] ist eine empfänger-initiierte, topologieunabhängige Lastverteilung für Multiprozessoren. Es wird sukzessive ein globales Potentialfeld für die Nähe der unterbelasteten Prozessoren approximiert. Die Lasteinheiten wandern von den überbelasteten Prozessoren in Richtung des Gradienten zu den schwach belasteten Prozessoren. Da die Lasteinheiten über mehrere Prozessoren wandern und nicht von ihnen als Last aufgenommen werden, kann das Gradienten-Modell nicht als strikt lokal bezeichnet werden. Außerdem ist eine Anwendung des Gradienten-Modells in Domänen mit stark dynamischen Belastungen (wie z.B. B&B) problematisch, da dort die Approximation des Potentialfelds zu schnell veraltet.

2.3 Baumsuche auf SIMD-Rechnern

Die heutigen SIMD-Rechner verfügen über sehr viele, oft mehrere 10.000 Prozessoren. Daher ist bei den (zumeist synchronen) Algorithmen zur Baumsuche nur eine vertikale Parallelisierung möglich. Mit einer solchen massiv-parallelen Architektur wird die zentrale Verwaltung der Datenstruktur OPEN in der horizontalen Parallelisierung zu einem indiskutablen Engpaß (Kapitel 1.3). Bei den Ansätzen der vertikalen Parallelisierung wird hier eine weitere Unterscheidung bezüglich der OPEN-Menge getroffen. Sie betrifft die Sortierung der Knoten in der verteilten Datenstruktur, die bei der Verfolgung einer bestimmten Strategie (z.B. Bestensuche oder Tiefensuche) notwendig ist. Diese Sortierung kann entweder global über alle verteilten Teilmengen hinweg oder lokal innerhalb jeweils einer Teilmenge vorgenommen werden. Im folgenden werden beide Varianten zur synchronen Baumsuche mit vertikaler Parallelisierung vorgestellt.

2.3.1 Lokal verwaltete Datenstruktur

Hier werden synchrone Ansätze zur Baumsuche beschrieben, deren Datenstruktur OPEN auf die Prozessoren verteilt und dort lokal sortiert ist.

In [Frye90] wird eine vollständige Tiefensuche für das Conway-Puzzle und den Baskett-Benchmark auf der Connection Machine CM-2 berechnet. Die Lastverteilung wird durch den statischen Trigger S^x (Kapitel 2.3.1) initiiert. Experimentell wird ein Wert von $x = 2/3$ als beste Schwelle für die meisten Rechnerkonfigurationen mit unterschiedlich vielen Prozessoren bestimmt. Für die Lastverteilung kommen zwei Ansätze zum Einsatz: Scan-directed-load-balancing (SDLB) bzw. Dimensions-Austausch (siehe Kapitel 2.1.3 bzw. 2.2.2). Anstatt beim SDLB den Fluß durch lokale Operationen zu bewerkstelligen, wird hier der globaler Router der Connection Machine für den Lasttransfer eingesetzt, wobei die belasteten Prozessoren Teile ihrer Arbeit an wartende Prozessoren abgeben. Andererseits wird beim Dimensions-Austausch nach einer reinen Load-sharing-Strategie vorgegangen, d.h. eine Balance der Last findet nicht statt. Die Leistung beider Lastverteilungen sind ähnlich, was dadurch begründet wird, daß die Lastverteilung vor allem in der Aufbauphase (Kapitel 1.3) notwendig ist und dort den gleichen Aufwand benötigt.

In [Mahanti93, Powley93, Karypis92] wird eine parallele Variante des Iterative-deepening-A* (IDA*) nach [Korf85] zur Lösung des 15-Puzzles auf der Connection Machine CM-2 untersucht. Das IDA*-Verfahren führt iterativ mehrere Tiefensuchen bis zu einer Schwelle in der Knotenbewertung durch und aktualisiert danach die Schwelle. In [Powley93] wird eine Lastverteilung für IDA* zu drei Zeitpunkten durchgeführt: Während der Aufbauphase, zwischen den Iterationen und während einer Iteration. Die Lastverteilung während einer Iteration entspricht der des B&B und ist laut Experimenten am wichtigsten. Als Lastverteilung wird hier die Rendezvous-allocation-method (Kapitel 2.1.2) angewendet.

Zu welchem Zeitpunkt die Lastverteilung eingesetzt wird, hängt von den in Kapitel 2.1.1 beschriebenen Trigger-Mechanismen ab. Falls der Trigger feuert, werden in [Powley93, Karypis92] durch Rendezvous-allocation die wartenden Prozessoren den beschäftigten zugeordnet und jeweils die Hälfte der Arbeit transferiert. Im Gegensatz zu diesem einmaligen Transfer wird in [Mahanti93] mehrmals Arbeit transferiert, aber jeweils nur ein Knoten übertragen. Außerdem wird neben der schwachen und starken Belastung ein Lastzustand "mittlere Belastung" verwendet. Durch diesen iterativen Ansatz kann eine optimale Balance der Last erreicht werden. Ein Vergleich dieser Lastverteilungsmethoden und weiteren Varianten wird in [Stalp93] für die B&B-Anwendung durchgeführt.

Unter den asynchronen, sender-initiierten Lastverteilungen für die Baumsuche gibt es eine Variante, die auch bei der SIMD-Verarbeitung anwendbar ist. Dabei werden nach jeder Knotenexpansion durch einen Prozessor die erzeugten Nachbarn an andere Prozessoren verteilt. Zur Bestimmung der empfangenden Prozessoren sind zwei Alternativen bekannt. Einerseits werden

in [Felten88] die Prozessoren zufällig ausgewählt und damit ein B&B mit Bestensuche auf einer asynchrone Hyperkubus-Struktur abgearbeitet. Andererseits können die empfangenden Prozessoren zyklisch durch "Multiplex-round-robin" ausgewählt werden [Huang89]. Bei insgesamt P Prozessoren schickt der Prozessor i den x-ten Nachfolger an den Prozessor $(i + x) \bmod P$. Diese Variante wird für den parallelen, iterativen A* für eine Menge von Prozessor-Speicher-Paaren mit $O(\log P)$ Zeitaufwand bei Interprozessor-Kommunikation vorgeschlagen. Für große Probleme wird in diesem Modell eine Beschleunigung von $O(P / \log P)$ berechnet. Die Qualität der dadurch entstehenden Lastverteilung hängt stark vom Grad der heuristischen Beschneidung des Suchbaums (effektiver Verzweigungsfaktor) ab.

2.3.2 Global verwaltete Datenstruktur

Bei der Baumsuche mit vertikaler Parallelisierung ist zwar die Datenstruktur OPEN auf die Prozessoren verteilt, aber die Knoten in den lokalen OPEN-Mengen können dennoch einer globalen Ordnung unterworfen sein. Realisierungen und Ergebnisse zu diesem Ansatz werden im folgenden beschrieben.

Eine Möglichkeit ist, die Knoten global nach deren Index im beschnittenen Suchbaum zu ordnen [Dehne90]. Dabei wird nicht der vollständige Suchbaum, sondern nur die aktiven Knoten in OPEN mit allen ihren Vorgängern betrachtet. Ein Knoten des beschnittenen Suchbaums erhält den Index i nach der natürlichen Numerierung[7] und wird in dem Prozessor i gespeichert. Nach jeder Knotenexpansion und Beschneidung des üblichen B&B werden die Indizes neu berechnet und die generierten Nachfolger auf die entsprechenden Prozessoren transferiert. Bei m Knoten in OPEN sind dazu asymptotisch $O(\log m)$ Schritte in einem Hyperkubus notwendig. In [Dehne90] wird angenommen, daß jeder Prozessor nur einen Knoten speichert. Diese Annahme ist nicht notwendig, falls die Knotenindizes modulo der Prozessoranzahl P gerechnet werden. Damit erreicht man eine optimale Lastverteilung, andererseits ist ein sehr hoher Kommunikationsaufwand nach jeder Iteration erforderlich. Experimentelle Ergebnisse liegen nicht vor.

Eine weitere Möglichkeit ist die Zuordnung der Knoten auf die Prozessoren mittels einer Hash-Funktion [Evett90]. Nach einer parallelen Knotenexpansion werden sukzessive die Indizes der erzeugten Nachfolger berechnet und die Nachfolger an die entsprechenden Prozessoren transferiert. Dieser Mechanismus wurde in dem Parallel-retraction-A* eingesetzt. Die dabei verwendete Hash-Funktion wird nicht angegeben. Sobald genug Knoten erzeugt wären, um alle Prozessoren zu beschäftigen, würde die Prozessorauslastung nahe bei 100 % sein, so daß die Anzahl der Iterationen klein ist. Dennoch ist nach jeder Iteration

---

[7] Natürliche Numerierung von Suchbäumen: Von der Wurzel beginnend, ebenenweise, von links nach rechts.

globale Kommunikation für den Knotentransfer notwendig, was sich im Zeitaufwand niederschlägt.

Prinzipiell ist auch eine globale Sortierung der OPEN-Menge nach den heuristischen Bewertungen der Knoten denkbar. Dies hätte z.B. den Vorteil, daß neben der Lastverteilung zusätzlich eine globale Suchstrategie realisiert werden könnte. Es konnten aber keine Arbeiten mit diesem Ansatz gefunden werden.

2.4 Schlußfolgerungen

Aus dem Überblick über den Stand der Forschung lassen sich einige Schlußfolgerungen ziehen, die hier mit den Bemerkungen aus der Einleitung zusammengefaßt werden sollen. Der grundlegende Ansatz dieser Arbeit ist, daß zur Beschleunigung des aufwendigen Optimierungsalgorithmus B&B massive Parallelität eingesetzt werden muß und diese heute vor allem durch feinkörnige SIMD-Rechner zur Verfügung gestellt wird. Eine der wichtigsten Eigenschaften massiv-paralleler Algorithmen ist ihre Skalierbarkeit, welche durch eine strikte Lokalität aller Operationen gewährleistet wird.

Das Hauptproblem bei dem feinkörnig parallelisierten B&B ist die Lastverteilung. Speziell für SIMD-Rechner wurden bisher globale oder semi-lokale Lastverteilungen untersucht. Auch bei den Triggern sind nur globale Mechanismen betrachtet worden. Diese Ansätze sind insofern sinnvoll, als daß die im Moment verfügbaren SIMD-Rechner, wie z.B. die MasPar MP-1 oder die Connection Machine CM-2, über leistungsstarke, hardware-gestützte Router verfügen. Leider läßt sich die Leistungsfähigkeit dieser Hardware nicht im gleichen Maße beliebig steigern, wie die Rechenkapazität durch Hinzunahme weiterer Prozessoren daher ist der Einsatz von lokalen Lastverteilungsmethoden in Zukunft notwendig. Andererseits wurden noch keine strikt lokalen Methoden zur Lastverteilung für SIMD-Rechner entworfen. Aus diesem Grund wird in Kapitel 4 eine solche Lastverteilung als wichtige Neuerung vorgeschlagen und untersucht.

Die bekannten lokalen Lastverteilungsverfahren haben einen für die Anwendung wesentlichen Nachteil: In allen drei vorgestellten Verfahren wird von einer kontinuierlichen Belastung der Prozessoren ausgegangen. Tatsächlich handelt es sich bei den meisten Anwendungen um diskrete Lasteinheiten. Insbesondere bei dem hier betrachteten B&B stellen die einzelnen Knoten des Suchbaums die Lastelemente dar. Andererseits kann bei massiv-parallelen Rechnern nicht zu einer kontinuierlichen Darstellung übergegangen werden, da der lokale Speicher relativ klein und damit die Anzahl der Lastelemente beschränkt ist. Eine kontinuierliche Lastdarstellung vereinfacht zwar die Analyse der Verfahren, führt aber zu der in Kapitel 2.2.1 beschriebenen Unausgewogenheit der Belastung. Daher ist es wichtig, Lastverteilungsverfahren zu

entwerfen, die dieses Problem berücksichtigen und z.B. von diskreten Lasteinheiten ausgehen.

Für die asynchrone Verarbeitung in MIMD-Rechner gibt es schon eine Reihe lokaler Lastverteilungsmethoden. Sie lassen sich auf den Diffusionsansatz zurückführen. Die Nachbarschafts-Mittelung ist von den Diffusionsansätzen für einige Topologien wie z.B. für bestimmte *K*-fache *D*-Kuben optimal. Sie benötigt bei wachsender Prozessoranzahl asymptotisch einen quadratischen Aufwand. Da sie von den bekannten lokalen Lastverteilungen die beste Methoden darstellt, wird sie in Kapitel 4 als Vergleich herangezogen.

Bei den bisherigen Untersuchungen von Lastverteilungsmethoden wurde als Aufwandsmaß ausschließlich die Anzahl der Kommunikationsschritte bzw. Lasttransfers verwendet. Für MIMD-Rechner ist dies ein adäquates Aufwandsmaß, da es sich oft um lose-gekoppelte Rechner handelt, bei denen die Aufsatzzeit der Kommunikation relativ groß zu der eigentlichen Übertragungszeit der Information ist. Dagegen ist bei eng-gekoppelten Rechnern dieses Verhältnis umgekehrt. Zur analytischen Bestimmung des Arbeitsaufwandes der synchronen Lastverteilung ist es daher wichtig, die übertragene Informationsmenge mit zu berücksichtigen. Die Menge ist auch immer größer oder gleich der Anzahl von Transfers, da für die Kommunikation von einer Informationseinheit mindestens ein Aufsetzen der Verbindung notwendig ist. Aus diesem Grund wird hier für den parallelen B&B auf SIMD-Rechner statt der Anzahl von Kommunikations*schritten* die Kommunikations*menge* betrachtet (Kapitel 4).

Betrachtet man die Baumsuche speziell für die feinkörnige SIMD-Verarbeitung, dann existieren nur wenige Arbeiten, welche die damit verbundenen Konsequenzen ausnutzen. Zu den Eigenschaften, welche SIMD-Rechner von anderen Architekturen abgrenzt, gehören neben der synchronen Verarbeitung, die enge Kopplung der Prozessoren sowie ein effizienter Broadcast- und Reduktionsmechanismus. Als einen Ansatz, der diese Eigenschaft ausnutzt, kann man die implizite Lastverteilung in Form der k-Expansion in Kapitel 5 ansehen.

Schließlich wird in der Literatur kaum auf die Aspekte der Aufbauphase des parallelen B&B eingegangen. Zumeist handelt es sich in der Regel um einen direkt von der sequentiellen Implementierung übernommenen Ansatz. Was dennoch durch anfängliches Abarbeiten des Suchbaums an statischer Lastverteilung beim parallelen B&B möglich ist, wird im folgenden Kapitel 3 untersucht.

3. Statische Lastverteilung

In diesem Kapitel werden Methoden zur statischen Lastverteilung bei parallelen Branch-and-bound-Algorithmen (B&B) beschrieben und untersucht. Die Aufgabe und die Notwendigkeit für die Betrachtung der statischen Lastverteilung wird in der Einleitung in Kapitel 3.1 motiviert. Dann folgt die Beschreibung von fünf unterschiedlichen Methoden und ihre Laufzeitmodellierung in Kapitel 3.2. Die Vorzüge und Nachteile der Methoden arbeitet ein qualitativer und analytischer Vergleich der Methoden in Kapitel 3.3 heraus. Schließlich wird der Einfluß der statischen Lastverteilung auf den nachfolgenden Algorithmus durch die experimentellen Ergebnisse in Kapitel 3.4 bestätigt.

3.1 Einleitung

Ein bei parallelen B&B-Algorithmen in der Aufbauphase (Kapitel 1.3) auftretendes Problem ist typisch für die parallele Bearbeitung von Suchbäumen, insbesondere wenn sie während der Bearbeitung generiert werden. Unter der Annahme, daß eine vertikale Parallelisierung auf Knotenebene vorgenommen wird, können nicht alle Prozessoren von Beginn an ausgenutzt werden. Der Suchbaum wird Knoten für Knoten expandiert, und die anstehende Arbeit wächst mit jeder Iteration. Beginnt man mit einem initialen Knoten (der Wurzelknoten des Suchbaums), dann wird die Anzahl der in der nächsten Iteration zu betrachtenden Knoten mit dem Verzweigungsfaktor b des Suchbaums multipliziert. Die Anzahl der Knoten und damit die Anzahl der aktiven Prozessoren kann mit der Tiefe d des Suchbaums höchstens mit b^d wachsen. Die Effizienz des gesamten Algorithmus wird reduziert, da nicht alle Prozessoren von Beginn an eingesetzt werden können.

Die Reduktion der Effizienz in der Aufbauphase ist eine problem-inhärente Eigenschaft. Zudem kann sie nicht durch eine perfekte Prozessorauslastung nach einigen Iterationen kompensiert werden. Dieses Problem tritt selbst bei idealisierten Parallelrechner-Modellen, wie z.B. der Parallel random access machine (PRAM) auf. Für reale Parallelrechner verstärkt sich dieses Problem

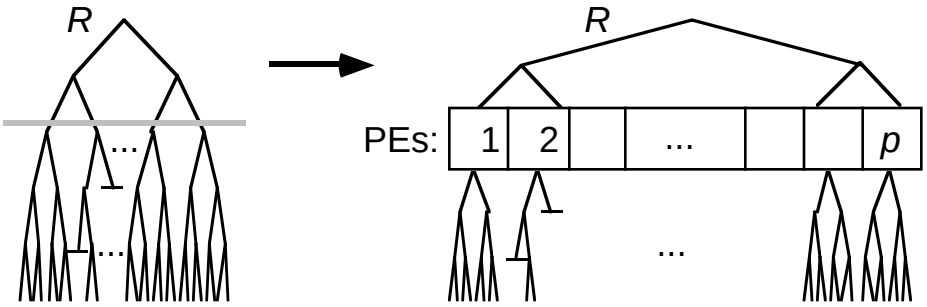

Abb. 3.1: Aufgabe der statischen Lastverteilung mit Wurzelknoten R *des Suchbaums und* P *Prozessorelementen (PEs)*

durch die Kommunikation, die zur Verteilung der Knoten auf die Prozessoren notwendig ist. Außerdem ist nach [Frye90] eine Lastverteilung vor allem in der Aufbauphase wichtig, da dort der größte Bedarf dafür bestehe (Kapitel 2.3.1). Schließlich wird durch eine ideale statische Lastverteilung die nachfolgende dynamische Lastverteilung im Prinzip überflüssig. Aus diesen Gründen ist es interessant, eine mögliche statische Lastverteilung des parallelen B&B zu untersuchen.

Auch bei der statischen Lastverteilung werden die zwei Ziele Load-balancing und Load-sharing verfolgt (Kapitel 1.4). Bei dem ersten soll die Lastverteilung vor Beginn des eigentlichen Algorithmus alle Prozessoren mit Arbeit versorgen. Beim zweiten soll die zukünftige Arbeit möglichst gleichmäßig auf die Prozessoren verteilt werden. Da bei der parallelen Abarbeitung von Suchbäumen die zukünftige Arbeit nicht bekannt ist, kann es sich bei dem zweiten Ziel nur um eine Annäherung handeln. Aus diesem Grund wird im folgenden von der "Initialisierung" des parallelen B&B statt der statischen Lastverteilung gesprochen.

Die hier untersuchten Ansätze zur Initialisierung des parallelen B&B basieren auf einer geschickten Erzeugung und Verteilung der Knoten des Suchbaums (Abb. 3.1). In Algorithmus 1.1 geschieht dies in der Funktion Init_Branch_and_bound() (Kapitel 1.2). Anstatt den Suchbaum sukzessive aufzubauen und die Verteilung der Knoten der dynamischen Lastverteilung zu überlassen, wird jedem einzelnen Prozessor wenigstens ein Knoten des Suchbaums zugewiesen. Alle für die Initialisierung verwendeten Knoten haben eine Mindesttiefe im Suchbaum, die durch die Anzahl von Prozessoren und dem Verzweigungsfaktor des Suchbaums bestimmt wird. Die Anzahl generierter Knoten sollte größer oder gleich der Prozessoranzahl sein, so daß jeder Prozessor mit mindestens einem Knoten versorgt wird. Falls mehr Knoten als Prozessoren in dieser Tiefe existieren, dann übernehmen einige Prozessoren mehrere Knoten, um die Korrektheit des Algorithmus zu gewährleisten. Nach der Initialisierung verwendet jeder Prozessor seine (lokale) Wurzel und fährt mit dem üblichen B&B fort. Insgesamt beginnt der B&B nach der statischen Lastverteilung seine

übliche Suche ab einer bestimmten Tiefe des Suchbaums statt von dem Wurzelknoten.

Zur Realisierung der statischen Lastverteilung kann eine Reihe von unterschiedlichen Methoden angegeben werden. Die wichtigsten davon werden in [Henrich93] beschrieben und ausgewertet. Der Zustand des Suchprozesses bzw. die Verteilung der Knoten auf die Prozessoren nach der Initialisierung ist je nach Methode unterschiedlich. Die Gewährleistung eines identischen Zustands nach verschiedenen Initialisierungen würde zwar den Vergleich der Methoden vereinfachen, erbringt aber bezüglich der Effizienz der Initialisierung oder des nachfolgenden Algorithmus keine Vorteile.

3.2 Methoden zur Initialisierung

In diesem Unterkapitel werden fünf verschiedene Varianten zur Initialisierung des parallelen B&B-Algorithmus beschrieben und ihre Laufzeiten modelliert. Da bisher die Initialisierungsmethoden weder für sich untersucht noch untereinander verglichen wurden, werden hier Namen zur Bezeichnung der Methoden eingeführt (Wurzel-, enumerative, selektive, direkte und erweiterte direkte Initialisierung). Für die formale Modellierung des Zeitaufwands der verschiedenen Initialisierungsmethoden werden für die Grundoperationen des parallelen B&B folgende Laufzeiten angenommen:

T_{send} sei die Zeit für das Versenden eines Knotens von dem Vorrechner an alle Prozessoren (Broadcast).

T_{gen} sei die Zeit für die Generierung von einem Nachfolger eines Knotens durch die Expansionsfunktion.

T_{eval} sei die Zeit für die Berechnung der unteren Schranke für die Kosten eines Knotens durch die Bewertungsfunktion (Evaluierung).

T_{trans} sei die Zeit für den Transfer eines Knotens zwischen zwei direkt benachbarten (verbundenen) Prozessoren.

Der Zeitaufwand für Operationen auf der Datenstruktur OPEN, die von einigen Verfahren verwendet werden, fällt nicht ins Gewicht und kann der Evaluierungszeit T_{eval} zugeschlagen werden. Nun folgen einige Definitionen, die die heuristische Beschneidung der Knoten im Suchbaum mit einbeziehen:

w_i aus dem Intervall [0, 1] sei der mittlere Anteil aller unbeschnittenen Knoten x in der Tiefe[1] i, also: $g(x) < f(z)$ (Kapitel 1.2)

[1] Die Beschneidung mit diesem Faktor kommt im B&B erst in der darauffolgenden Tiefe $i + 1$ zum Tragen, denn die Knoten in Tiefe i müssen erzeugt und evaluiert werden, bevor sie beschnitten werden können.

$\bar{w} = \frac{1}{d} \sum_{i=1}^{d} w_i$ sei der mittlere Anteil der unbeschnittenen Knoten (Beschneidungsgrad).

d_w sei die kleinste Tiefe des Suchbaums mit $w_{d_w} \geq P$.

3.2.1 Wurzelinitialisierung

Bei der Wurzel-Initialisierung (*root initialization*, RI) wird die Wurzel des B&B-Suchbaums in eine der lokalen OPEN-Teilmengen eingefügt. Gemäß der Hauptschleife des sequentiellen B&B wird der Wurzelknoten von dem Prozessor entnommen und expandiert. Die Nachfolger der Wurzel werden dann nach dem dynamischen Lastverteilungsschema verteilt, welches auch in der Bearbeitungsphase des B&B angewendet wird. Die anderen Prozessoren, die einen Knoten erhalten, verfahren auf die gleiche Weise. Nach einer bestimmten Zeitdauer sieht die gesamte OPEN-Menge für jeden Prozessor zumindest einen Knoten vor, und der B&B geht von der Aufbauphase in die Bearbeitungsphase über. Abgesehen von der dynamischen Lastverteilung, läßt sich die RI durch Abb. 3.2 illustrieren.

Die Anwendung der RI ist in Beschreibungen des parallelen B&B ein weit verbreiteter Ansatz. Dies kann dreierlei Ursachen haben: (1) Entweder die Initialisierung ist nicht das Hauptanliegen der Beschreibung, (2) es wird ein Berechnungsmodell verwendet, welches keine Kommunikation zwischen den Prozessoren berücksichtigt (z.B. PRAM-Modell) oder (3) die Initialisierung wurde direkt von der sequentiellen Implementierung des B&B übernommen.

Die RI ist sehr einfach zu realisieren, da sie die Lastverteilungsmethode des Hauptalgorithmus verwendet und keinen zusätzlichen Mechanismus benötigt. Andererseits wird im Vergleich zu anderen Initialisierungen eine kostenintensive Verteilung der Nachfolgerknoten verwendet. Jede Verteilung ist mit dem dazu notwendigen Kommunikationsaufwand verbunden. Weiterhin sind viele Prozessoren während der Aufbauphase unbeschäftigt und warten darauf, einen Knoten zu erhalten.

Unter der Annahme, daß der B&B einen Suchbaum mit konstantem Verzweigungsfaktor bearbeitet, wird nun die zur Initialisierung notwendige Anzahl von Iterationen betrachtet. Die Aufbauphase ist beendet, wenn alle Prozessoren zumindest einen Knoten erhalten haben. Der Zeitbedarf dieser Phase hängt logarithmisch von der Anzahl von Prozessoren ab. Wenn ein Suchbaum mit Verzweigungsfaktor b mit P Prozessoren verarbeitet wird, dann können alle Prozessoren frühestens nach $\log_b(P)$ Iterationen einen Knoten erhalten. Ob alle Prozessoren tatsächlich einen Knoten zugewiesen bekommen, hängt von dem konkreten Mechanismus zur dynamischen Lastverteilung ab.

Für die Modellierung der Laufzeit wird angenommen, daß durch die dynamische Lastverteilung eine perfekte Verteilung der Knoten vorgenommen wird. Dann ergibt sich der Zeitaufwand T_{RI} aus mehreren Termen. Zum einen

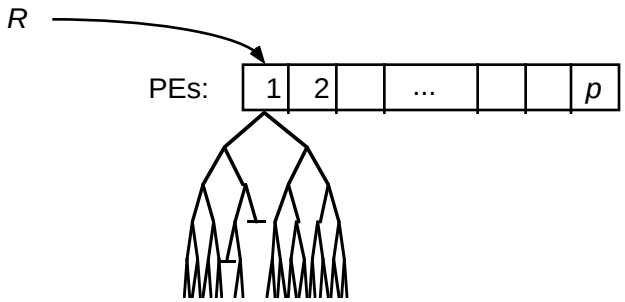

Abb. 3.2: Bearbeitungsschema der Wurzelinitialisierung (RI) mit Wurzelknoten R *des Suchbaums und* P *Prozessorelementen (PEs)*

wird der Wurzelknoten des Suchbaums von dem Vorrechner auf einen beliebigen Prozessor in der Zeit T_{trans} transferiert. Der Prozessor berechnet dann in der Zeit T_{eval} die Bewertung der Wurzel. Danach werden bis zur (a priori unbekannten, aber festen) Tiefe $d_w \geq \log_b(P)$ von allen Prozessoren mit nicht-leerer OPEN-Menge folgende drei Schritte durchgeführt:

(1) Expandieren des aktuellen Knotens und Generieren der im Mittel b Nachfolgerknoten in jeweils der Zeit T_{gen}.

(2) Evaluierung der b generierten Knoten in jeweils der Zeit T_{eval}. Falls die untere Schranke eines Knoten x schlechter ist als die erste heuristische Lösung, dann wird x nicht weiter betrachtet. In der Tiefe i wird also nur noch der Teil w_i aller Knoten behalten.

(3) Transferieren der im Mittel $w_i \cdot b$ Nachfolger zu benachbarten Prozessoren, die noch über keinen Knoten verfügen in der Zeit T_{trans}.[2]

Insgesamt ergibt sich eine zusammengesetzte Zeit, die zum einen die Behandlung der Wurzel und zum anderen die aufsummierten Iterationenszeiten beinhaltet:

$$T_{\text{RI}} = T_{\text{trans}} + T_{\text{eval}} + \sum_{i=1}^{d_w} b\,(T_{\text{gen}} + T_{\text{eval}} + w_i\,T_{\text{trans}})$$

Mit dem mittleren Beschneidungsgrad $\overline{w}$ ergibt dies für die RI folgende Modellierung des Zeitverhaltens:

$$T_{\text{RI}} = T_{\text{trans}} + T_{\text{eval}} + d_w\,b\,(T_{\text{gen}} + T_{\text{eval}} + \overline{w}\,T_{\text{trans}}) \qquad (3.1)$$

3.2.2 Enumerative Initialisierung

Die zweite Methode zur Initialisierung (*enumerative initialization*, EI) ist sehr ähnlich zur vorherigen und wird für die B&B-Algorithmen in

[2] Da man nicht annehmen kann, daß die emfangenden Prozessoren immer in direkter Nachbarschaft zum sendenden Prozessor liegen, stellt die konstante Zeit T_{trans} eine untere Abschätzung dar. Eine genauere Abschätzung ist von der zugrunde liegenden Verbindungsnetzwerk der Prozessoren abhängig.

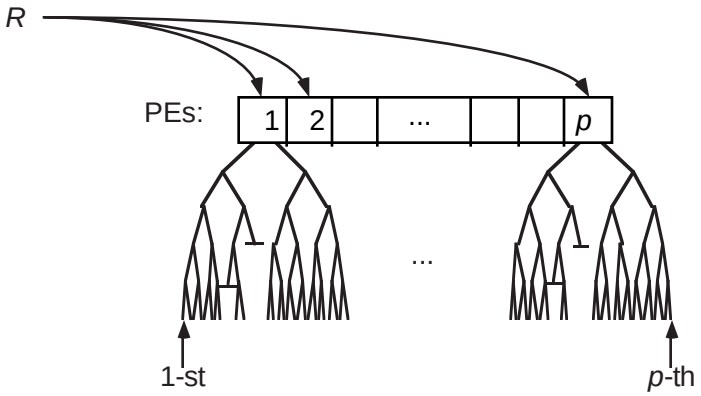

Abb. 3.3: Schema für die enumerative Initialisierung (EI) mit Wurzelknoten R *des Suchbaums und* P *Prozessorelementen (PEs)*

[Vornberger86, Pargas88] verwendet. Der wesentliche Unterschied liegt in dem Versenden desselben Wurzelknotens an alle Prozessoren zu Beginn der Initialisierung. Danach ist bei jedem Prozessor derselbe Knoten in der lokalen OPEN-Menge enthalten. Dieser Knoten und seine Nachfolger werden gemäß der Hauptschleife des sequentiellen B&B von jedem Prozessor expandiert. Alle *P* Prozessoren schreiten solange fort, bis sie mindestens *P* Knoten in ihrer OPEN-Menge erzeugt haben. Dann behält der *i*-te Prozessor den *i*-ten Knoten der Menge und verwirft die restlichen (Abb. 3.3). Mit diesem eindeutigen Knoten führen die Prozessoren den lokalen B&B durch und der gesamte B&B-Algorithmus wechselt von der Aufbauphase zur Bearbeitungsphase. Insgesamt werden also die für die Initialisierung der Prozessoren notwendigen Knoten von jedem einzelnen Prozessor enumeriert.

Neben der bisher beschriebenen Initialisierung ist eine Strategie notwendig, welche die Korrektheit des Algorithmus garantiert. Das Problem tritt vor allem dann auf, wenn in dem letzten Schritt der Aufbauphase mehrere Knoten generiert werden. In diesem Fall würde die obige Initialisierung die überflüssigen Knoten verwerfen, die möglicherweise zu der einzigen optimalen Lösung führen. Eine zulässige Methode verteilt nach dem letzten Schritt die überflüssigen Knoten an die anderen Prozessoren. Dabei werden die empfangenden Prozessoren gemäß ihren Indizes der Reihe nach zyklisch ausgewählt (round robin).

Die Anzahl der Iterationen in der Aufbauphase der EI ist ähnlich zu der von RI. Um mindestens *P* Knoten generieren zu können sind ungefähr $\log_b(P)$ Iterationen notwendig, falls ein konstanter Verzweigungsfaktor *b* angenommen wird. Obwohl gegenüber der RI keine Verbesserung in der Anzahl von Iterationen erreicht wird, hat die EI Vorteile. Falls der Suchbaum in der Aufbauphase von allen Prozessoren expandiert wird, dann kann jeder Prozessor seine "lokale Wurzel" ohne weitere Kommunikation generieren. Alle Prozessoren sind von Anfang an beschäftigt und müssen nicht warten, bis sie einen Knoten erhalten. Andererseits wird redundante Arbeit geleistet, da viele

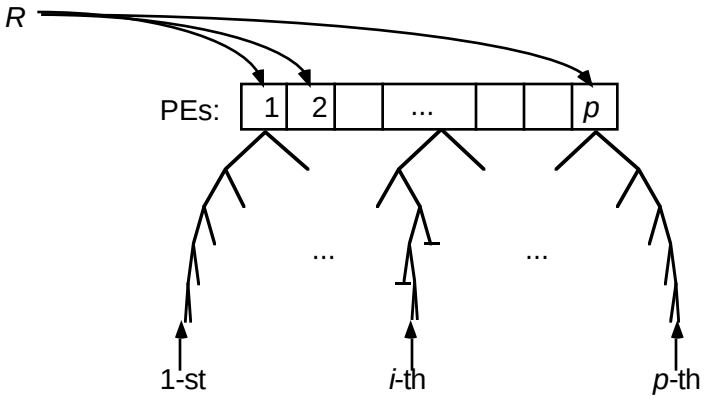

Abb. 3.4: Schema für die selektive Initialisierung (SI) mit Wurzelknoten R *des Suchbaums und* P *Prozessorelementen (PEs)*

Prozessoren die gleichen Knoten bearbeiten und identische Auswertungen durchführen. Außerdem ist relativ viel lokaler Speicher pro Prozessor notwendig.

Im Zeitverhalten unterscheidet sich die EI von der RI durch das Versenden der Wurzel an alle Prozessoren in der Zeit T_{send}. Die Evaluierung der Wurzel bleibt dabei gleich. Dann werden von jedem Prozessor bei Breitensuche bis zur Tiefe d_w die Knoten des Suchbaums generiert und evaluiert ($T_{\text{gen}} + T_{\text{eval}}$). Es werden jedoch in der Tiefe i nur diejenigen Knoten weiterverfolgt, die einen nicht beschnittenen Vorgänger in Tiefe $i-1$ haben. Daraus resultiert der Faktor w_{i-1} in der Zeitabschätzung T_{EI}, mit:

$$T_{\text{EI}} = T_{\text{send}} + T_{\text{eval}} + \sum_{i=1}^{d_w} b^i \, w_{i-1} \, (T_{\text{gen}} + T_{\text{eval}}) \qquad (3.2)$$

3.2.3 Selektive Initialisierung

Die selektive Initialisierung (*selective initialization*, SI) ist eine Methode mit Modulo-Arithmetik, die zuerst von [ElDessouki80] eingeführt und später auch in [Ma88] benutzt wurde. Dabei werden die Nachteile der EI durch eine lokale Auswahlstrategie der Knoten auf den einzelnen Prozessoren vermieden.

Die Methode beginnt, analog zu der EI, mit dem Versenden des Wurzelknotens an alle Prozessoren. Anstatt nun den gesamten Suchbaum bis zu einer bestimmten Tiefe zu generieren, beschreitet jeder Prozessor nur einen einzelnen Pfad im Suchbaum. Um diesen Pfad abzugehen führt jeder Prozessor eine Reihe von Expansions- und Selektionsschritten durch. Dabei wird der aktive Knoten (zu Beginn die Wurzel) expandiert und seine Nachfolger generiert. Statt alle Nachfolger in der OPEN-Menge zu speichern, wird nur einer weiterverfolgt. Die vollständigen Pfade der Prozessoren sind verschieden und führen zu einem geeigneten Knoten für die Initialisierung der Prozessoren (Abb. 3.4).

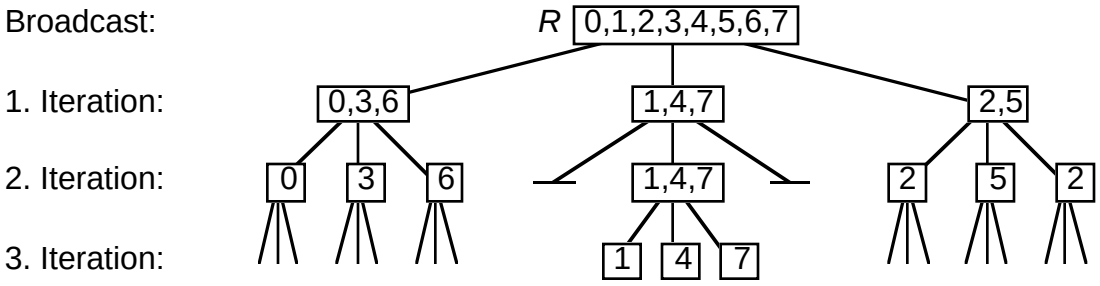

Abb. 3.5: Beispiel für die selektive Initialisierung von P = *8 Prozessoren durch einen Suchbaum mit Wurzel R und Verzweigungsfaktor 3. Die Ziffern zu den Suchbaumknoten geben die Indizes* pe_id *der Prozessoren an, welche den entsprechenden Knoten in ihrer lokalen OPEN-Menge speichern.*

Die jeweilige Auswahl des nächsten aktiven Knotens aus der Reihe von Nachfolgern wird durch die Modulo-Berechnungen aus [ElDessouki80] bewerkstelligt. Dabei hat jeder Prozessor, der an dem B&B-Algorithmus teilnimmt, einen eindeutigen Index *pe_id* zur Identifikation. Die Anzahl der aktuellen Prozessoren, welche denselben bisher zurückgelegten Pfad beschritten haben, wird durch *ncp* bezeichnet. Wenn zusätzlich *size* die Anzahl von generierten Nachfolger angibt, dann wählt der Prozessor *pe_id* den *g*-ten Knoten mit

$$g = ((pe\_id - 1) \text{ MOD } size) + 1 \tag{3.3}$$

In der nächsten Iteration werden die eingeführten Variablen gemäß den folgenden drei Zuweisungen aktualisiert:[3]

$$delta = \begin{cases} 1, & g \le ncp \text{ MOD } size \\ 0, & \text{ansonsten} \end{cases} \tag{3.4}$$

$$pe\_id = \lceil pe\_id \ / \ size \rceil \tag{3.5}$$

$$ncp = \lfloor ncp \ / \ size \rfloor + delta \tag{3.6}$$

Die Schleife terminiert, wenn *size* größer oder gleich *ncp* wird. Dann erhält jeder Prozessor $\lfloor size / ncp \rfloor$ Stück der Nachfolgerknoten. Der letzte Prozessor übernimmt die restlichen Knoten. Ein Beispiel für die Zuordnung der Knoten zu den Prozessoren durch diesen Mechanismus ist in Abb. 3.5 gegeben.

Verglichen mit der RI hat diese Methode den gleichen Vorteil wie EI: Es ist nur wenig Kommunikation notwendig. Außerdem haben die Prozessoren weniger redundante Arbeit, denn nur der erste Teil des Pfades im Suchbaum eines Prozessors ist identisch mit dem eines anderen Prozessors. In jeder Iteration werden die Prozessoren eines Pfadstückes auf die nachfolgenden Zweige aufgeteilt. Aus diesem Grund ist die Effizienz dieser Initialisierung größer als bei RI und EI. Auf der anderen Seite erfolgt die Auswahl der Knoten nicht nach

---

3 Die Berechnung von *delta* ist eine korrigierte Version der Formel in [ElDessouki80].

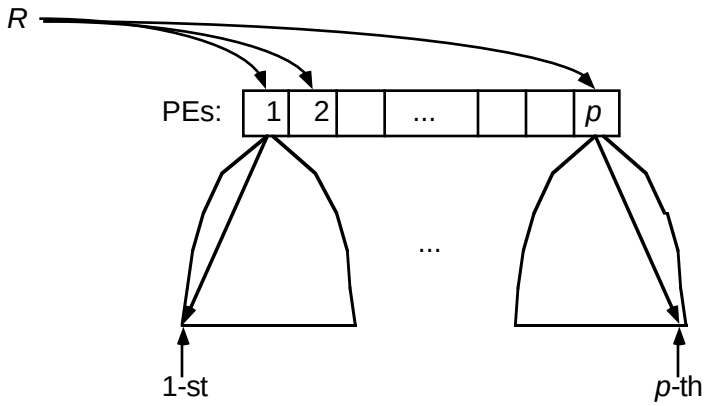

Abb. 3.6: Schema für die direkte Initialisierung (DI) mit Wurzelknoten R *des Suchbaums und* P *Prozessorelementen (PEs)*

einer Bestensuche. Im Gegensatz zu dem sequentiellen B&B werden hier die Knoten mehr oder weniger unabhängig voneinander generiert. Die Knoten werden weder nach ihrer heuristischen Bewertung sortiert noch verglichen. Insgesamt werden die *P* Prozessoren nicht mit den *P* besten Knoten initialisiert.

In dieser zirkulären Methode wird nichts über den Fall gesagt, falls *size* zu Null wird. Dies geschieht, wenn keine Nachfolger der Knoten existieren oder alle erzeugten Nachfolger durch die derzeit beste Lösung (obere Schranke) beschnitten werden. Weiterhin wird keine Lastverteilung in der Aufbauphase angewendet. Daher werden die Prozessoren ohne Nachfolger unbeschäftigt, bis sie in die Bearbeitungsphase eintreten und durch die Lastverteilung Arbeit erhalten. Insgesamt hat diese Methode etwas mehr Wartezeiten als die EI.

Bezüglich der Zeitmodellierung von SI entspricht das Versenden des Wurzelknotens dem Vorgehen von EI. Jedoch wird der Baum selbst von mehreren Prozessoren parallel bis zur Tiefe $d_w - 1$ abgearbeitet. Dabei werden in jeder Iteration nacheinander b Nachfolger generiert und ausgewertet. Der für diese Initialisierung benötigte Zeitaufwand T_{SI} entspricht:

$$T_{\mathrm{SI}} = T_{\mathrm{send}} + T_{\mathrm{eval}} + d_w\, b\, (T_{\mathrm{gen}} + T_{\mathrm{eval}}) \tag{3.7}$$

3.2.4 Direkte Initialisierung

Die direkte Initialisierung (*direct initialization*, DI) wurde zuerst in [Vornberger87] für das Vertex-Cover-Problem verwendet. Eine sequentielle Version dieser Initialisierung wird in [Abdelrahman88] erwähnt, ohne die mögliche parallele Verarbeitung auszunutzen.

Bei der DI wird der Suchbaum nicht explizit abgearbeitet, sondern die entsprechenden Knoten direkt erzeugt. Dazu muß die Struktur des Suchbaums bekannt sein, Das heißt, der i–te Knoten in der Tiefe d_0 kann direkt, ohne seine Vorgänger zu betrachten, berechnet werden. Damit kann jeder Prozessor seinen lokalen (Wurzel-) Knoten als Initialisierung in annähernd konstanter Zeit generieren (Abb. 3.6). Da jeder Prozessor mit unterschiedlichen Knoten versorgt

werden sollte, werden bei der Initialisierung alle Knoten in einer bestimmten Tiefe d_0 des Suchbaums erzeugt. Der Parameter d_0 wird dabei so gewählt, daß der Suchbaum in dieser Tiefe mindestens P Knoten enthält. Für ein festen Verzweigungsfaktor b berechnet sich d_0 aus $\lceil \log_b(P) \rceil$. Falls in dieser Tiefe mehr Knoten als Prozessoren existieren, dann müssen einige Prozessoren mehrere Knoten übernehmen, um die Korrektheit des Verfahrens zu bewahren. Nach der Initialisierung fährt jeder Prozessor mit dem üblichen parallelen B&B fort, nur daß nun in einer bestimmten Tiefe statt von der Wurzel begonnen wird.

Dieser Ansatz zur Initialisierung beinhaltet die wichtigsten der Vorteile der letzten beiden Methoden (EI, SI), Das heißt, wenig Kommunikation und wenig Wartezeiten. Zusätzlich führen die Prozessoren keine redundante Arbeit aus. Andererseits werden mögliche ungünstige Zweige des Suchbaums nicht durch die derzeit beste Lösung beschnitten. Dies ist darin begründet, daß während der Initialisierung keine heuristische Bewertung der Knoten durchgeführt wird. Die Beschneidung der Knoten ist am Ende der Aufbauphase möglich und führt unter Umständen zu unbeschäftigten Prozessoren. Der zweite Nachteil ist der gleiche wie bei der SI: Es werden nicht die P besten Knoten zur Initialisierung ausgewählt.

Wie die anderen Methoden versendet und evaluiert auch die DI zunächst die Wurzel. Dann werden jedoch die Knoten in Tiefe d_0 direkt in einem Schritt generiert und evaluiert. Dieser Generationsschritt entspricht zwar nicht exakt dem bisherigen, ist aber zeitlich vergleichbar. Falls $P < b^{d_0 - 1}$ gilt, dann müssen in Tiefe d_0 je nach Verzweigungsfaktor b mehrere Knoten pro Prozessor berechnet werden. Damit ergibt sich für die DI ein Zeitaufwand T_{DI} von:

$$T_{\mathrm{DI}} = T_{\mathrm{send}} + T_{\mathrm{eval}} + \left\lceil \frac{b^{d_0}}{P} \right\rceil (T'_{\mathrm{gen}} + T_{\mathrm{eval}}) \tag{3.8}$$

3.2.5 Erweiterte direkte Initialisierung

Bei der obigen direkten Initialisierung wird versucht, jeden Prozessor mit mindestens einem Knoten zu versorgen. Dieses Vorgehen entspricht einem Load-sharing. Die hier diskutierte erweiterte direkte Initialisierung (*extended direct initialization*, EDI) geht darüber hinaus in die Richtung Load-balancing, wenn auch die Last, die ein einzelner Knoten darstellt, nicht ausgewertet wird. Bei der EDI werden absichtlich mehrere Knoten pro Prozessor erzeugt, denn nach Simulationsergebnissen aus [Suttner93] resultiert dies in einer besseren Lastverteilung.

Sei für die EDI die maximale Anzahl s von Knoten, die pro Prozessor erzeugt werden sollen, gegeben. Nach dem Versenden der Wurzel an alle Prozessoren, generiert dann die Initialisierung bei konstantem Verzweigungsfaktor b alle Knoten aus der Tiefe $d_s = \lfloor \log_b(P \cdot s) \rfloor$ (Abb. 3.7). Falls die lokalen OPEN-Mengen der Größe l möglichst vollständig gefüllt werden sollen, dann muß $s = l$ gewählt werden.

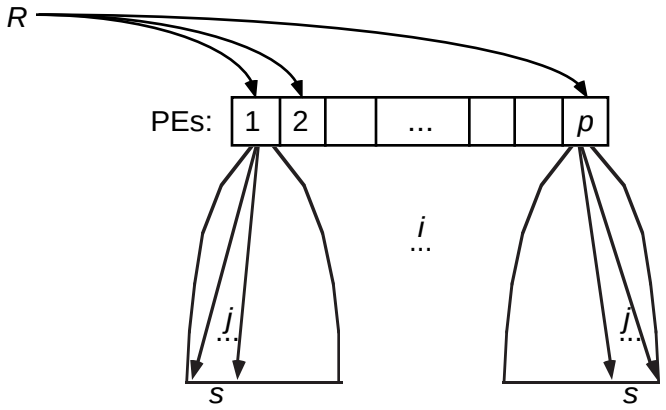

Abb. 3.7: Schema für die erweiterte direkte Initialisierung (EDI) mit Wurzelknoten R *des Suchbaums und* P *Prozessorelementen (PEs) zur Generierung von* s *Knoten pro Prozessor*

Nach der Wahl der Tiefe d_s ist bei der EDI noch die Zuordnung der einzelnen Knoten auf die Prozessoren offen. Diese Zuordnung wird durch eine diskrete, bijektive Funktion

$$Z(i, j) \text{ mit } Z\text{: } \{0, \ldots, P-1\} \times \{0, \ldots, l-1\} \rightarrow \{0, \ldots, l \cdot P - 1\} \tag{3.9}$$

festgelegt. Sie gibt für das j-te Element von OPEN des i -ten Prozessors den Index des Knotens in der Tiefe d_s an. Mit der natürlichen Numerierung des gesamten Suchbaums[4] ergibt dies den Knoten mit der Nummer $b^{d_s-1} + Z(i, j)$. Falls $Z(i, j) > b^{d_s}$ gilt, dann existieren keine Knoten mehr in Tiefe d_s und der Index $Z(i, j)$ wird als leeres Element interpretiert. Die einfachste Zuordnung von Knoten auf Prozessoren ist Z_1 mit:

$$Z_1(i, j) = i \cdot l + j \tag{3.10}$$

Dabei werden die Knoten der Reihe nach den ersten Prozessoren zugeordnet, bis ihre lokalen Speicherresourcen (OPEN-Mengen) erschöpft sind. Mit den anderen OPEN-Mengen wird genauso verfahren, bis keine Knoten in der Tiefe d_s mehr verfügbar sind. Die restlichen Prozessoren bekommen keine Knoten zugeordnet. Eine andere Zuordnung Z_2 füllt die OPEN-Mengen gleichmäßig an. Dazu werden den OPEN-Mengen der Reihe nach nur ein Knoten zugewiesen. Am Ende sind gegebenenfalls einige OPEN-Mengen nicht ganz gefüllt, aber die Größe der OPEN-Mengen unterscheidet sich höchstens um einen Knoten. Die Zuordnungsfunktion Z_2 ist gegeben durch:

$$Z_2(i, j) = j \cdot P + i \tag{3.11}$$

Allgemein kann für die Zuordnungsfunktion $Z(i, j)$ eine beliebige bijektive Abbildung gewählt werden. Die verschiedenen Zuordnungen unterscheiden sich bezüglich der Qualität der Lastverteilung. Da nach der Erzeugung und Bewertung

---

[4] Natürliche Numerierung von Suchbäumen: Von der Wurzel beginnend, ebenenweise, von links nach rechts.

der Knoten einige Knoten beschnitten werden, drückt sich die Qualität in der Verteilung der verbleibenden Knoten aus. Daher bezeichnen wir bei der EDI eine Lastverteilung als gut, falls die nicht beschnittenen Knoten möglichst gleichmäßig auf die Prozessoren verteilt werden. Nun kann nicht im voraus bestimmt werden, welcher Knoten beschnitten wird und welcher nicht. Dennoch kann man durch die Wahl der Zuordnungsfunktion die Wahrscheinlichkeit der Knotenverteilung beeinflussen: Bei der erweiterten direkten Initialisierung (EDI) ist die Zuordnung $Z_2(i, j)$ optimal in dem Sinne, daß die Wahrscheinlichkeit für eine gleichmäßige Verteilung der nicht beschnittenen Knoten auf die Prozessoren im Rahmen einer gleichverteilten Beschneidung maximal ist.

Der Grund für die Optimalität von $Z_2(i, j)$ ist folgender: Vor der Beschneidung sind durch die Zuordnung $Z_2(i, j)$ die Knoten gleichmäßig auf die Prozessoren verteilt, das heißt, die Größe der OPEN-Mengen unterscheidet sich höchstens um einen Knoten. Falls alle Vorgänger y in einer Tiefe $d < d_s$ günstige Bewertungen haben $(g(y) < f(z)$, mit der derzeit besten Lösung z), dann kann die Beschneidung von Knoten nicht vorhergesagt werden. Andernfalls gibt es zu einem ungünstigen Knoten x in Tiefe d_s auch einen ungünstigen Vorgänger y in einer Tiefe $d < d_s$. Auf Grund der Monotonie der Bewertungsfunktion haben auch alle anderen Nachfolger von y in Tiefe d_s eine ungünstige Bewertung. Durch die hier gewählte Knotennumerierung sind alle diese Nachfolger direkt benachbart. Das heißt, die Wahrscheinlichkeit einen unbeschnittenen Knoten z in der Nachbarschaft von x zu finden, wächst mit der Distanz zwischen x und z. Andererseits können bei P Prozessoren maximal P direkt benachbarte Knoten unterschiedlichen Prozessoren zugewiesen werden. Somit ist der Abstand zwischen zwei Knoten, die demselben Prozessor zugewiesen wurden, maximal P, und gerade dies leistet die Zuordnung $Z_2(i, j)$.

Für die EDI ist die Zeitmodellierung analog zur DI. Nur werden hier pro Prozessor nicht nur ein Knoten sondern bis zu l Knoten erzeugt. Dabei ist der Parameter l durch die Größe von OPEN nach oben beschränkt und d_l berechnet sich aus $d_l = \lfloor \log_b(P \cdot l) \rfloor$. Damit ist der Zeitaufwand T_{EDI} der EDI gegeben durch:

$$T_{\text{EDI}} = T_{\text{send}} + T_{\text{eval}} + \left\lceil \frac{b^{d_l}}{P} \right\rceil (T_{\text{gen}} + T_{\text{eval}}) \tag{3.12}$$

Durch die Erweiterung der EDI wird der Nachteil der DI noch verstärkt. Das heißt, die Teilbäume werden noch später beschnitten (erst in Tiefe d_s statt in Tiefe d_0). So entsteht ein Effizienzverlust, denn einige Prozessoren müssen ungünstige und damit unnötige Knoten erzeugen und auswerten. Durch Beschneidung ihres gemeinsamen Vorgängers würde sich die Arbeit auf einen Knoten reduzieren. Andererseits werden die Knoten in der Tiefe d_s schnell und ohne Kommunikation erzeugt. Die Knoten in geringeren Tiefen müssen dabei nicht betrachtet werden. Dadurch daß mehrere Knoten pro Prozessor erzeugt werden, ist bei optimaler Zuordnung auch nach der Beschneidung eine gute Verteilung

der Last wahrscheinlich. Bei der einfachen DI müssen Beschneidungen durch eine dynamische Lastverteilung ausgeglichen werden.

Offen bleibt die Frage, ob die verbesserte statische Lastverteilung die späte Beschneidung ausgleicht und insgesamt eine Beschleunigung des Gesamtalgorithmus erbringt. Diese Frage wird in einem späteren Unterkapitel durch experimentelle Ergebnisse beantwortet.

3.3 Vergleich der Methoden

Bei dem Einsatz des parallelen B&B für eine konkrete Anwendung stellt sich die Frage, welche Initialisierung am günstigsten ist. Abhängig von der Problemstellung und der Rechnerarchitektur können die verschiedenen Methoden sehr unterschiedliche Laufzeiten und Auswirkungen auf den nachfolgenden Algorithmus haben. Zur Bewertung der einzelnen Initialisierungsmethoden werden in den nachfolgenden Unterkapiteln Kriterien herangezogen und die Laufzeiten der Methoden bzw. die Wartezeiten der Prozessoren verglichen.

3.3.1 Qualitativer Vergleich

Die verschiedenen Methoden werden in Tabelle 3.1 anhand mehrerer Kriterien verglichen. Die Ausprägungen aller ausgewählten Kriterien haben einen nicht zu vernachlässigenden Einfluß auf die einzelnen Methoden.

- Das erste Kriterium zeigt die Menge der notwendigen *Kommunikation* während der Aufbauphase zwischen den Prozessoren an.
- Nur falls die Prozessoren unterschiedliche Knoten bearbeiten, wird der potentielle Parallelismus ausgenutzt. Daher ist die *Redundanz* der durchgeführten Arbeit ein wichtiges Kriterium. Es wird gemessen in der Anzahl mehrfach generierter und damit mehrfach bearbeiteter Knoten.
- Die *Wartezeiten* der Prozessoren dienen als Indikator für eine Abnahme der Effizienz. Sie tauchen in unserem Algorithmus dann auf, wenn ein Prozessor keine Arbeit hat und warten muß bis er von einem anderen Prozessor einen Knoten erhält.
- Die *Allgemeinheit* einer Initialisierungsmethode ist dann gegeben, falls sie auf alle diskreten Optimierungsprobleme, die durch den B&B-Algorithmus gelöst werden, anwendbar ist. Wie die Tabelle zeigt, haben nicht alle Methoden diese Eigenschaft.
- Der *variable Verzweigungsfaktor* zeigt an, ob eine Methode Suchbäume mit nicht konstantem Verzweigungsfaktor behandeln kann und ob einige Zweige wegen ihrer ungünstigen Kostenabschätzung beschnitten werden können.

	Wurzel-Init. (RI)	Enumerat. Init. (EI)	Selektive Init. (SI)	Direkte Init. (DI)	Erweiterte Init. (EDI)
wenig Kommunikat.	–	+	+	+	+
wenig Redundanz	+	–	±	+	+
wenig Wartezeiten	–	+	±	+	+
Allgemeinheit	+	+	+	–	–
variabler Verz.faktor	+	+	+	–	–
Bestensuche	+	+	–	–	–
Lastverteilung	+	+	+	–	±

Tabelle 3.1: Vergleich der fünf vorgestellten Methoden zur Initialisierung bzw. statischen Lastverteilung für paralleles Branch-and-bound anhand qualitativer Kriterien

- Das vorletzte Kriterium beschreibt, ob die am Ende der Aufbauphase ausgewählten Knoten der globalen *Bestensuche* genügen, das heißt, die *P* Prozessoren die *P* besten Knoten erhalten.
- Das letzte Kriterium *Lastverteilung* gibt an, wie gleichmäßig die erzeugten Nachfolgerknoten auf die Prozessoren verteilt werden.

In Tabelle 3.1 gibt eine Bewertung mit drei Ausprägungen (+, ±, –) an, inwieweit ein Kriterium auf eine Methode relativ zu den anderen Methoden zutrifft (gut, mittelmäßig, schlecht). Um eine konsistente Auswertung zu ermöglichen sind die ersten drei Kriterien in ihrer negierten Form angegeben. Betrachtet man zum Beispiel die Redundanz, dann werden die RI und die DI als gut, die SI als mittelmäßig und die EI als schlecht bezüglich der Vermeidung von redundanter Arbeit ausgewertet.

Aus der Bewertung der Initialisierungen in Tabelle 3.1 können für die drei allgemeinen Verfahren zwei Schlußfolgerungen gezogen werden. Einerseits ist bei ausreichend großem Speicher bzw. wenigen Prozessoren die EI gegenüber der RI zu bevorzugen, da sie bis auf die Redundanz mehr Vorteile erbringt. Die Redundanz ist hier aber zu vernachlässigen, da während den redundanten Berechnungen keine anderen Knoten zur Verfügung stehen, die bearbeitet werden könnten. Wenn man andererseits auf die Bestensuche verzichtet, da z.B. eine Tiefensuche durchgeführt wird, dann ist die SI zu bevorzugen. Bei minimaler Kommunikation wird dort die geringste redundante Arbeit geleistet.

Ist die Struktur des Suchbaums bekannt, dann kommen auch die letzten beiden Initialisierungen in Frage. Sie entsprechen bezüglich dem Kommunikationsaufwand der SI, schlagen diese aber bezüglich der Redundanz und den Wartezeiten. Dem gegenüber steht die fehlende Möglichkeit zur Beschneidung ungünstiger Knoten. Die EDI kann gegenüber der direkten durch Erzeugung mehrerer Knoten pro Prozessor eine bessere Verteilung der Nachfolger erbringen. Welche der drei verbleibenden Initialisierungen günstiger ist, zeigt sich im Vergleich der Laufzeiten und später in den Experimenten.

3.3.2 Analytischer Vergleich

In Kapitel 3.2 wurde für die einzelnen Initialisierungsmethoden die Laufzeit modelliert. Die Laufzeit der Initialisierung ist vor allem für kleine und mittlere Probleme wichtig, da dort der Zeitanteil der Initialisierung relativ groß ist. Dagegen ist bei großen Problemen die Verteilung und Beschneidung der Knoten während der Initialisierung entscheidend. Hier wird nun versucht, den Zeitaufwand der verschiedenen Methoden zu vergleichen. Bis zu einem gewissen Detaillierungsgrad geschieht dies für allgemeine Rechnerarchitekturen. Danach wird die Modellierung auf SIMD-Architekturen eingeschränkt.

Für die RI und die SI kann kein abschließender Vergleich angestellt werden, denn die Differenzbildung von T_{RI} und T_{SI} aus den Gleichungen 3.1 und 3.7 zeigt eine Abhängigkeit der Initialisierungslaufzeiten von dem Kommunikationsaufwand bei der Sende- und Transferoperation:

$$T_{\mathrm{RI}} - T_{\mathrm{SI}} \;=\; (d_w\, b\, w + 1)\, T_{\mathrm{trans}} - T_{\mathrm{send}} \tag{3.13}$$

Für realistische Werte von T_{trans} und T_{send} ist diese Differenz sicherlich größer Null, so daß die SI in den meisten Fällen schneller ist. Die restlichen Vergleiche erbringen, bis auf die EDI, welche von dem Parameter s abhängt, eindeutige Ergebnisse. Ein Koeffizientenvergleich der in allen Modellierungen vorkommenden Zeitkonstanten ergibt folgende Relation:

$$T_{\mathrm{DI}} \;\leq\; T_{\mathrm{SI}} \;\leq\; T_{\mathrm{EI}} \tag{3.14}$$

Die DI benötigt den kleinsten Zeitaufwand. Dabei ist zu beachten, daß die SI und DI aufgrund von Beschneidungen keine vollständige Belegung aller Prozessoren garantieren kann.

Für die Einschränkung der Rechnerarchitektur für SIMD-Verarbeitung gelten die gleichen Annahmen und Definitionen wie in Kapitel 3.2. Zusätzlich kann für SIMD-Rechner angenommen werden, daß über einen Datenbus ein Knoten an alle Prozessoren in konstanter Zeit verschickt werden kann (unabhängig von P). Vereinfachend soll $T_{\mathrm{trans}} = T_{\mathrm{send}}$ gelten. Damit folgt aus Gleichung 3.13:

$$T_{\mathrm{SI}} \;<\; T_{\mathrm{RI}} \tag{3.15}$$

Die Schlußfolgerung aus diesem Vergleich ist, daß, auf die Laufzeit bezogen, die DI am schnellsten durchzuführen ist, gefolgt von der SI und der EI. Für SIMD-Rechner gilt zusätzlich, daß die SI schneller als die RI ist. Das heißt, selbst die idealisierte dynamische Lastverteilung kann eine explizite Initialisierung zeitlich nicht ersetzen.

3.4 Experimentelle Ergebnisse

In den letzten Kapiteln wurden die verschiedenen Methoden zur Initialisierung vorgestellt und analysiert. Bei einem analytischen Vergleich der Methoden wurden die heuristischen Beschneidungen nicht berücksichtigt, da die Heuristik nicht befriedigend modelliert werden kann. Daher sollen hier experimentelle Ergebnisse in einer konkreten Anwendungsdomäne als Vergleich herangezogen werden. Durch die Experimente soll geklärt werden, wie effektiv und effizient die einzelnen Initialisierungen sind und wie sie sich auf den nachfolgenden B&B-Algorithmus auswirken. Außerdem ist eine mögliche Abhängigkeit von der Problemgröße von Interesse.

Für alle folgenden Ergebnisse gelten die gleichen experimentellen Randbedingungen. Aus der Anwendungsdomäne der non-operationalen Belegungsplanung (Kapitel 1.5) dienen 30 Probleminstanzen aus dem Anhang in Kapitel 7.1 als ein Benchmark-Satz. Die Probleme sind je nach Anzahl expandierter Knoten in die drei Schwierigkeitsklassen *einfach, mittel* und *schwer* eingeteilt. Der verwendete Algorithmus ist ein paralleler Branch-and-bound mit einer Tiefensuche (Kapitel 1.2 und 1.3). Dabei wird zu Beginn der Suche eine erste Lösung heuristisch berechnet. Die angewendete Lastverteilung ist das Flüssigkeitsmodell LM-C5, welches noch in Kapitel 4.1 vorgestellt wird. Sie wird von einem periodischen Trigger P^f mit Frequenz $f = 1$ initiiert (Kapitel 4.4). Der zugrundeliegende SIMD-Rechner ist eine MasPar MP-1 mit 16.384 Prozessoren, angeordnet in einem 2-dimensionale Torus. Da die Experimente nur zu einem relativen Vergleich dienen, konnte mit einer nicht-optimierten Implementierung in der Programmiersprache MPL (Erweiterung der Sprache C) gerechnet werden.

Von den vorgestellten Methoden zur Initialisierung werden in den Experimenten alle Verfahren bis auf die EI eingesetzt. Die zur Verfügung stehende Rechnerplattform ist nur mit 16 KByte Speicher pro Prozessorelement ausgestattet. Davon muß ein Teil für das Laufzeitsystem reserviert werden. Bei einer Knotengröße von 236 Bytes können daher $l = 50$ Knoten in der lokalen OPEN-Menge gespeichert werden. Mit dieser Rechnerkonfiguration kann die EI nicht eingesetzt werden, da sie bei $P = 16.384$ Prozessoren nicht alle notwendigen P Knoten pro Prozessor erzeugen kann. Für die erweiterte direkte Initialisierung wird die Anzahl s zu erzeugender Knoten pro Prozessor auf $s = l = 50$ Nachfolger festgelegt.

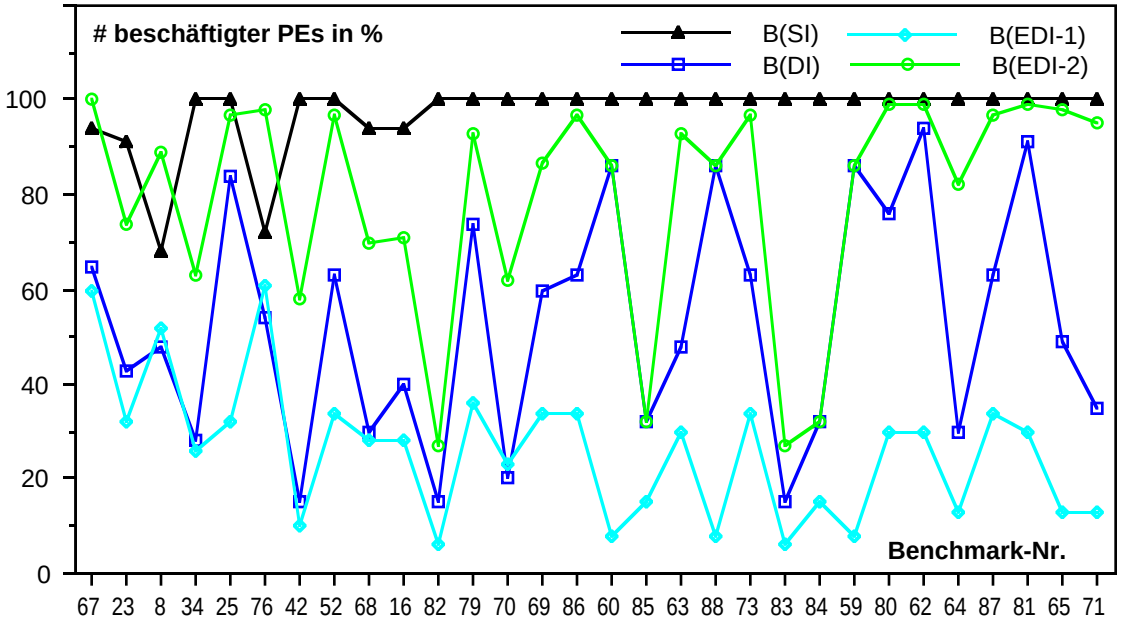

Abb. 3.8: Anteil beschäftigter Prozessoren (Load-sharing) nach Ausführung verschiedener Initialisierungsmethoden und Probleminstanzen in Prozent

3.4.1 Prozessorauslastung

Die Hauptaufgabe der Initialisierung ist die Versorgung der Prozessoren mit Knoten. Sie kann quantifiziert werden durch den prozentualen Anteil der Prozessoren, die nach der Aufbauphase mit mindestens einem Knoten versorgt sind. Dies ist ein Maß für die Effektivität der Initialisierung und daher ein wichtiges Vergleichskriterium. Die Abb. 3.8 zeigt die Effektivität als prozentualer Anteil beschäftigter Prozessoren für die verschiedenen Initialisierungen. Speziell die EDI ist für die in Kapitel 3.2.5 angegebenen Zuordnungsfunktionen Z_1 und Z_2 (fortan bezeichnet mit EDI-1 und EDI-2) aufgeteilt. Die Benchmark-Probleme sind nach der Problemgröße sortiert.

Für diese Anwendungsdomäne erbringt SI eine nahezu vollständige Versorgung der Prozessoren mit Knoten, vor allem für mittlere und große Probleme. Bei kleineren Problemen wird die SI teilweise von der EDI-2 übertroffen. Dies ist darin begründet, daß bei kleinen Problemen die Heuristik relativ zur Prozessoranzahl sehr früh die Knoten beschneidet. Die SI kann solchen Prozessoren keine Knoten mehr zuweisen, und sie bleiben bis zur dynamischen Lastverteilung unbeschäftigt. Dagegen findet bei der EDI-2 eine gute Durchmischung der zugewiesenen Knoten statt. Bis auf SI ist die Effektivität aller Initialisierungen über die Probleminstanzen sehr schwankend. Aber bei genauem Hinsehen, zeigt ein Vergleich der zwei EDI, daß die optimale Zuordnung Z_2 immer die bessere Knotenaufteilung als Z_1 erbringt. Die DI pendelt zwischen diesen beiden Werten.

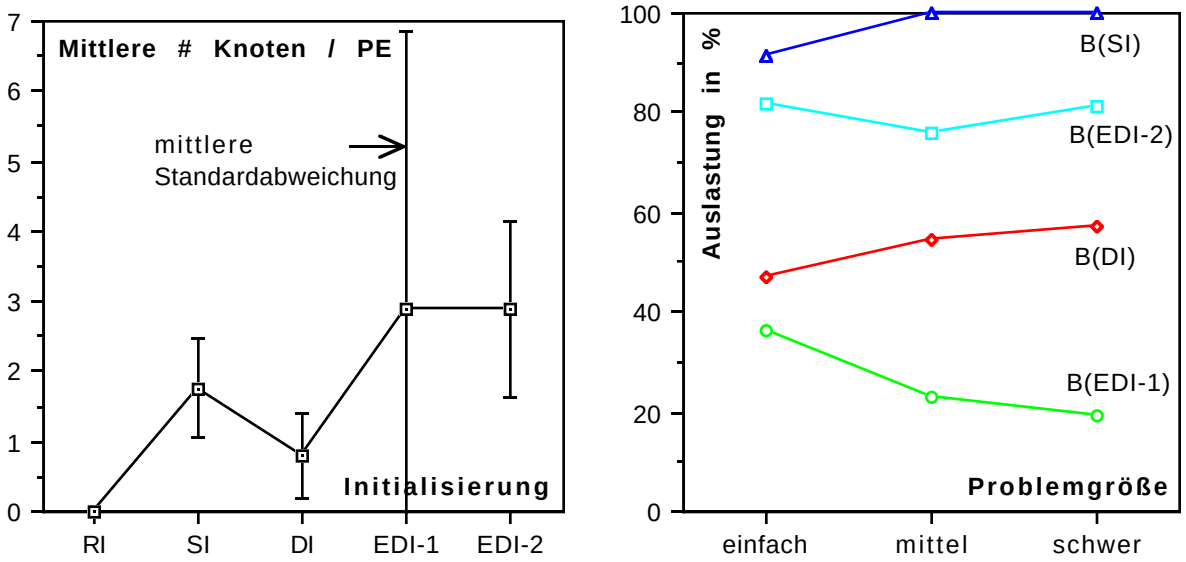

Abb. 3.9: Über alle Benchmark-Probleme gemittelte Versorgung der Prozessoren mit Knoten nach Ausführung der Initialisierung

In der Abb. 3.9 ist links die mittlere Anzahl $\overline{N}$ von erzeugten Knoten und die mittlere Standardabweichung $\overline{V}$ nach der Aufbauphase für die einzelnen Initialisierungsmethoden gegeben. Anstrebenswert ist ein hoher Mittelwert bei kleiner Standardabweichung, so daß möglichst viele Prozessoren ausgelastet sind (Mittelwert) und die Last möglichst gleichverteilt ist (Standardabweichung). Für die RI nähern sich diese Werte mit wachsender Anzahl von Prozessoren dem Wert Null. SI und DI haben kleine Mittelwerte und kleine Abweichungen. Bei der EDI werden mit beiden Zuordnungsschemata die gleiche (mittlere) Anzahl von Knoten erzeugt. Jedoch unterscheiden sie sich stark in der Verteilung $\overline{V}$ der Knoten. Mit der Zuordnung Z_2 erzielt EDI-2 ein weit besseres Ergebnis als EDI-1, was sich nach der Initialisierung auch in der Anzahl B beschäftigter Prozessoren aus Abb. 3.10 niederschlägt.

In dem rechten Teil der Abb. 3.9 ist die prozentuale Auslastung B über die Probleminstanzen aus jeweils einer Schwierigkeitsklasse gemittelt. Dies verdeutlicht nochmals die Rangfolge der Initialisierungsmethoden mit abnehmender Effektivität: SI, EDI-2, DI und EDI-1. Außerdem sind nur kleine Abhängigkeiten von der Problemgröße zu beobachten. Nach den Ergebnissen aus Abb. 3.9 ist SI auch die deutlich effizienteste Initialisierung. Dieser Schluß wird durch weitere Ergebnisse im folgenden Kapitel relativiert.

3.4.2 Gesamtlaufzeiten

Für eine abschließende Bewertung der verschiedenen Methoden sind die jeweiligen Laufzeiten des Gesamtalgorithmus mit den unterschiedlichen Initialisierungen von Bedeutung. In Abb. 3.10 sind sie in logarithmischem Maßstab für den gleichen Satz von Problemen wie oben angegeben. Jedoch sind

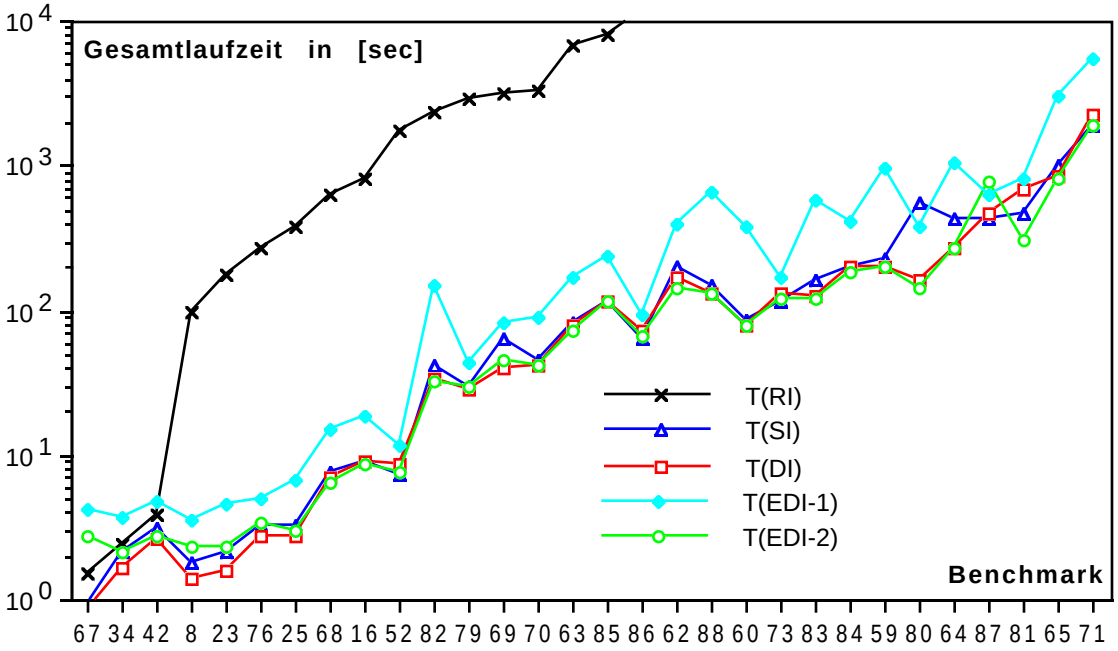

Abb. 3.10: Gesamtlaufzeiten des parallelen Branch-and-bound-Algorithmus beim Einsatz von verschiedenen Initialisierungsmethoden für den Satz der Benchmark-Probleme

sie hier zur besseren Übersicht nach den Laufzeiten steigend sortiert.[5] Da die Laufzeit nicht nur von der Problemgröße, sondern auch von der jeweiligen Probleminstanz abhängt, ergibt sich eine leicht veränderte Reihenfolge.

Die RI ist mit Abstand die ineffizienteste Initialisierungsmethode. Die resultierende Laufzeit ist um mehrere Größenordnungen schlechter als für die restlichen Initialisierungen. Aus diesem Grund konnten auch nur die 19 einfacheren Probleme berechnet werden. Die RI ist damit für die Baumsuche auf massiv parallelen Rechnern mit größerem Netzwerkdurchmesser ungeeignet. Die Laufzeiten der restlichen Initialisierungen bilden ein gleichmäßig anwachsendes Feld. Dabei führt der Einsatz von EDI-1 zu dem nächst langsameren Algorithmus. Bei den schwierigeren Problemen ist SI bzw. EDI-2 am günstigsten. Für einfachere Probleme hingegen ist DI vorzuziehen. Der Unterschied ist darin begründet, daß DI die ersten b^{d-1} Knoten nicht berechnet, sondern die Knoten in der Tiefe d direkt generiert. Damit ist die DI zwar schneller, aber es werden nicht alle Prozessoren mit Knoten versorgt. Dieser Nachteil wirkt sich erst bei schwereren Problemen aus.

In Abb. 3.11 sind links analog zu dem vorigen Unterkapitel die mittlere Gesamtlaufzeit $\overline{T}$ und die mittlere Anzahl $\overline{I}$ von Iterationen für die unterschiedlichen Initialisierungen angegeben. Die Anzahl von Iterationen zeigt beim Vergleich der verschiedenen Methoden ein sehr ähnliches Verhalten wie die Laufzeit. Für den gesamten Satz der Benchmark-Probleme, das heißt, für eine

[5] Als maßgebendes Sortierkriterium wurden die Laufzeiten der Wurzel- Initialisierung verwendet.

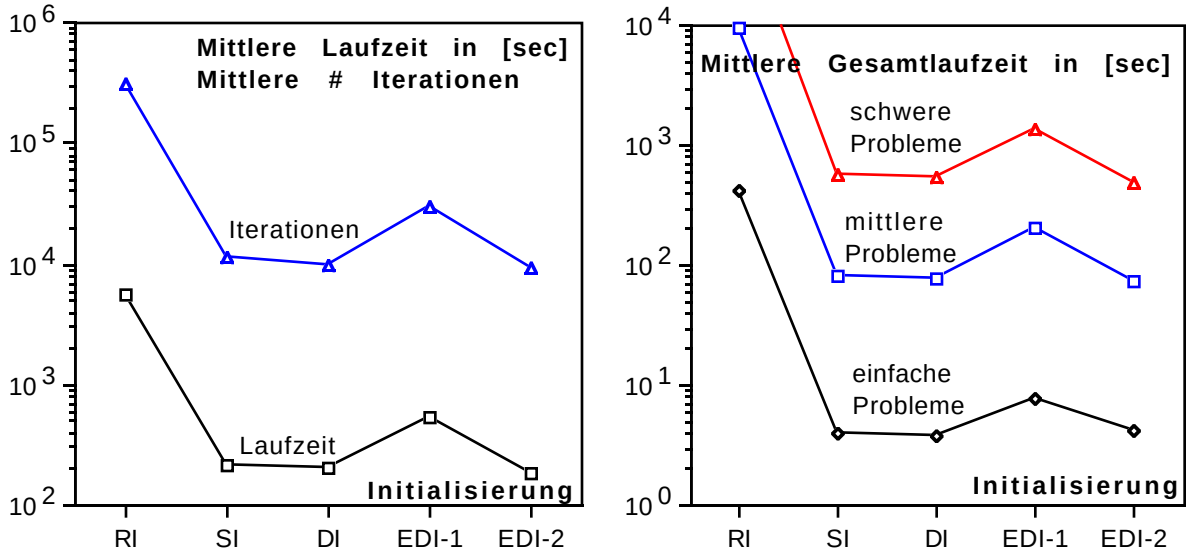

Abb. 3.11: Mittlere Gesamtlaufzeiten und Anzahl der Iterationen je nach Initialisierung und Schwere des Problems

Mischung aus einfachen, mittleren und schweren Problemen, ist die EDI-2 gefolgt von DI und SI die günstigste Initialisierung. Mit Abstand folgt dann die EDI-1. Zu indiskutablen Laufzeiten führt RI.

In dem rechten Diagramm der Abb. 3.11 sind die Gesamtlaufzeiten über die drei Problemklassen gemittelt. Zwischen den günstigsten Initialisierungen (SI, DI und EDI-2) sind nur marginale Unterschiede festzustellen. In der Tendenz ist EDI-2 für schwere und mittlere Probleme günstig, dagegen DI für kleine Probleme. Die RI ist trotz ihrer simplen Methodik selbst für die einfachen Probleme im Mittel nicht sinnvoll einzusetzen.

Die Wichtigkeit der Wahl der Initialisierungsmethode wird durch die Auswirkung ihre auf den nachfolgenden Algorithmus deutlich. In Abb. 3.12 sind die Differenzen der Laufzeiten von der Initialisierung an sich und des Gesamtalgorithmus gegenübergestellt. Die Differenzen wurden gegen die Laufzeit unter Einsatz der SI gebildet und jeweils für die drei Problemklassen in logarithmischen Maßstab aufgetragen. Zum Beispiel ist die Laufzeitdifferenz für einfache Probleme der RI gegenüber der SI bei der Initialisierung in der Größenordnung 10^0 (linke Säule) und bei dem Gesamtalgorithmus 10^3 (zweite Säule von links). Das heißt, der Einsatz von RI statt SI ergibt eine minimale Verlangsamung durch die veränderte Initialisierung, aber einen Laufzeit-differenz im Gesamtalgorithmus, die betragsmäßig drei Größenordnungen größer ist. Die Auswirkungen der Wahl von Initialisierungsmethoden geht überproportional in die Laufzeit des Gesamtalgorithmus ein. Dieser Effekt tritt bei mittleren und schwierigen Problemen verstärkt auf.

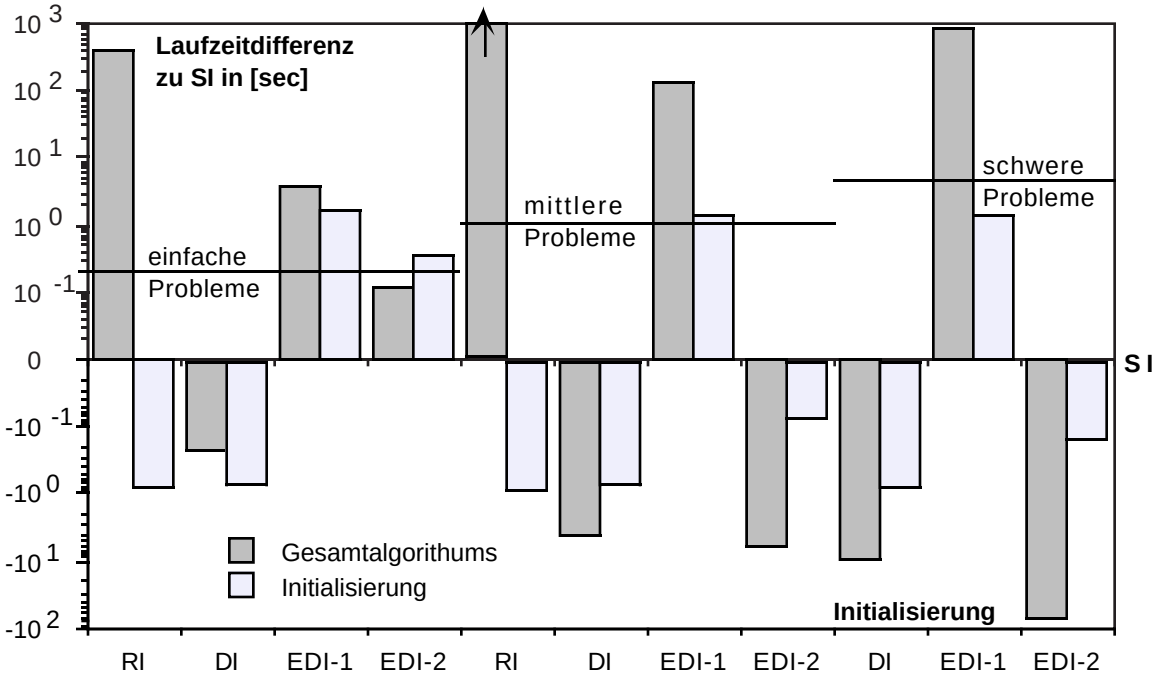

Abb. 3.12: Laufzeitdifferenzen der Initialisierungen und der Gesamtalgorithmen zur Verwendung der selektiven Initialisierung für alle Problemklassen

3.4.3 Zusammenfassung

Als Zusammenfassung sind in Tabelle 3.2 die über alle Probleme gemittelten Meßwerte angegeben. Diese Werte sollen keine absoluten Aussagen darstellen, sondern nur die Verfahren untereinander vergleichen. Angegeben sind Werte, die sich ausschließlich auf die Initialisierung beziehen, oder den Gesamtablauf des B&B berücksichtigen. Zu der Initialisierung gehören die mittlere Anzahl $\overline{B}$ beschäftigter Prozessoren nach der Initialisierung, die mittlere Anzahl $\overline{N}$ von Knoten pro Prozessor und ihre Standardabweichung $\overline{V}$. Zu den Werten bezüglich des Gesamtablaufs des B&B zählen die Laufzeit $\overline{T}$, die Anzahl $\overline{I}$ von Iterationen und die mittlere Summe $\sum I$ der Iterationen über alle Prozessoren.

Als Ergebnisse der Experimente lassen sich mehrere Punkte nennen, wobei zu berücksichtigen ist, daß diese Ergebnisse im Rahmen der verwendeten Anwendungsdomäne stehen. Zum einen ist die Effektivität der Initialisierungen, das heißt, die Auslastung der Prozessoren nach der Aufbauphase, sehr unterschiedlich. Selbst über die einzelnen Probleminstanzen ist die Auslastung durch die jeweilige Methode stark schwankend. Dies gilt nicht für die SI, denn sie erreicht für mittlere und schwere Probleme ein Auslastung von durchweg 100 %. Dennoch steht die Effektivität nicht in direktem Zusammenhang mit der Effizienz des Gesamtalgorithmus (gemessen durch die Laufzeit).

Insgesamt sind SI, DI und EDI-2 von allen betrachteten Initialisierungen am günstigsten. Falls die Struktur des Suchbaums bekannt ist, und damit alle drei verbleibenden Initialisierungen eingesetzt werden können, dann hängt die schlußendliche Wahl bei einem konkreten Anwendungsproblem von der Einfachheit der Implementierung ab. Im Gegensatz zu SI ist dies sicherlich bei der DI und der EDI-2 gegeben. Die RI ist dagegen für die Baumsuche mit lokaler, dynamischer Lastverteilung auf massiv-parallelen Rechnern mit größerem Netzwerkdurchmesser nicht sinnvoll einsetzbar.

	$\overline{B}$	$\overline{N}$	$\overline{V}$	$\overline{T}$	$\overline{I}$	$\sum I$
RI [6]	0,0 %	0,000	0,000	5.624,9 sec	310.964,4	$2{,}61 \cdot 10^7$
SI	97,1 %	1,760	0,694	212,97 sec	11.289,2	$12{,}74 \cdot 10^7$
DI	52,9 %	0,800	0,591	208,52 sec	9.780,0	$12{,}71 \cdot 10^7$
EDI-1	26,1 %	2,897	3,932	529,78 sec	29.273,1	$12{,}84 \cdot 10^7$
EDI-2	79,7 %	2,897	1,266	187,59 sec	9.369,7	$12{,}86 \cdot 10^7$

Tabelle 3.2: Über alle betrachteten Probleminstanzen gemittelten Meßergebnisse

Schließlich ist die Betrachtung der statischen Lastverteilung in Form von Initialisierungsmethoden für den massiv-parallelen B&B wichtig. Trotz kleiner Änderungen der Laufzeit durch die Initialisierung selbst werden große Veränderungen im nachfolgenden Algorithmus bewirkt. Diese Beeinflussung wird auch nicht von einer gleichzeitig zu dem B&B angewendeten lokalen Lastverteilung ausgeglichen.

6 Die Werte für RI beziehen sich nur auf 19 einfache bzw. mittlere Probleme des Benchmarks. Sie können daher nur als untere Schranke dienen.

4. Dynamische Lastverteilung

In diesem Kapitel wird ein neues dynamisches lokales Lastverteilungsverfahren vorgestellt und untersucht. Der Einsatzbereich beschränkt sich nicht nur den Branch-and-bound-Algorithmus [Henrich94a, Henrich94b], sondern erstreckt sich allgemein auf irreguläre Algorithmen [Henrich94c]. Diese Lastverteilung ist an ein physikalisches Modell angelehnt (Kapitel 4.1) und nützt die enge Kopplung der Prozessoren des verwendeten Rechnermodells aus. Mit Hilfe eines formalen Rahmen werden in Kapitel 4.2 die Eigenschaften und die Effizienz der Lastverteilung analysiert. Die aus dem Asynchronen übertragene Nachbarschafts-Mittelung dient zum Vergleich der Leistungsfähigkeit der Verfahren. Für einen besseren Einblick in das Verhalten der lokalen Lastverteilung wird in Kapitel 4.3 der Lastverteilungsprozeß von dem B&B-Prozeß abgetrennt und in einer Simulation untersucht. Die Frage, wann der Lastverteilungsprozeß fortgeführt werden soll, wird durch die Einführung eines lokalen Trigger-Mechanismus in Kapitel 4.4 beantwortet. Experimentelle Ergebnisse mit dem Gesamtalgorithmus in Kapitel 4.5 ermöglichen den Vergleich der Laufzeiten und damit eine Beurteilung der Leistungsfähigkeit.

4.1 Das Flüssigkeitsmodell

Zur Beschreibung der Methode wird zunächst die Idee anhand eines physikalischen Modells veranschaulicht. Dann folgt eine formale Darstellung des Algorithmus. Schließlich werden mittels eines Beispiel einige wichtige Effekte verdeutlicht.

4.1.1 Anschauung

Die vorgeschlagene lokale Lastverteilung implementiert ein physikalisch vereinfachtes Flüssigkeitsmodell (Liquid Model). In diesem Modell ist ein flacher Behälter mit homogener Flüssigkeit gefüllt. Im ausgeglichenen Zustand hat die Flüssigkeit an jedem Ort im Behälter die gleiche Höhe. Wenn nun an einem

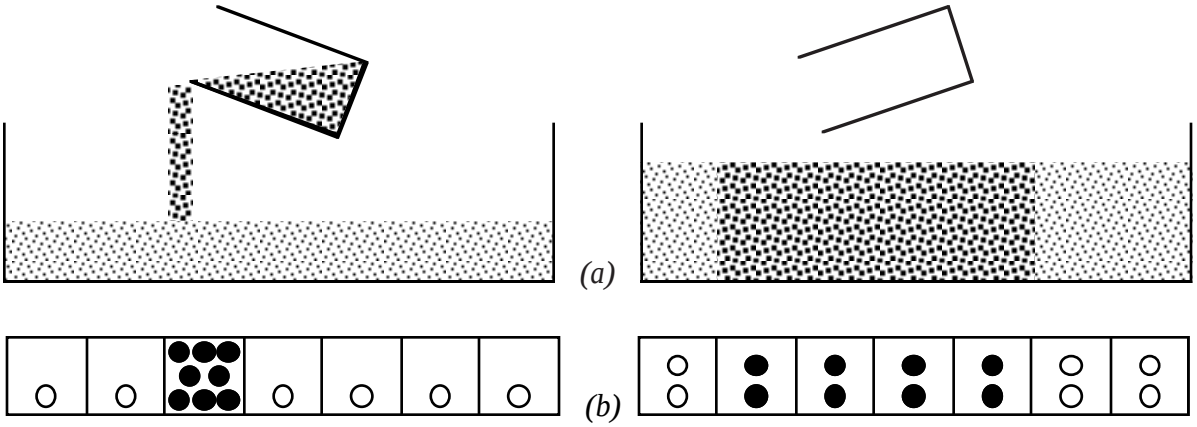

Abb. 4.1: Das Verhalten eines vereinfachten Flüssigkeitsmodells (a) und sein diskretes Äquivalent (b) bei neu hinzukommender Flüssigkeit bzw. Belastung

beliebigen Ort im Behälter lokal weitere Flüssigkeit hinzugegossen wird, dann gleicht sich der Flüssigkeitspegel im Behälter nach kurzer Zeit wieder aus. Abb. 4.1(a) verdeutlicht diesen Vorgang in vereinfachter Weise. Dieser Ausgleich geschieht aufgrund der Verdrängung der ursprünglichen Flüssigkeit durch die neu hinzukommende. Mittels sukzessivem Verdrängen wird insgesamt eine Gleichverteilung der Flüssigkeit geschaffen. Dieser globale Effekt kommt durch einen fortgesetzten streng lokalen Mechanismus zustande, da keines der Flüssigkeitsmoleküle zu einem Ort mit mangelnder Flüssigkeit "springen" wird.

Ein diskretes Äquivalent zu dem kontinuierlichen Modell ist für den eindimensionalen Fall in Abb. 4.1(b) angegeben. Dieser vereinfachte physikalische Effekt läßt sich auf die Lastverteilung übertragen. Dabei wird die Geometrie des Behälters diskretisiert und jedes Intervall entspricht einem Prozessor. Die Flüssigkeit im Flüssigkeitsmodell entspricht der Last im Lastverteilungsprozeß. Die Quantisierung der Flüssigkeitshöhe wird durch elementare Lasteinheiten repräsentiert. Im B&B-Algorithmus stellen die noch zu bearbeitenden Knoten bzw. Teilprobleme aus den lokalen OPEN-Mengen die Lasteinheiten dar.

Falls in irgendeinem Prozessor viel und in einem anderen Prozessor wenig Last aufkommt, dann sollte die Arbeit von dem ersteren zum letzteren Prozessor transferiert werden. Globale Ansätze würden diese Prozessoren lokalisieren und die Last direkt über ein Kommunikationsnetzwerk transferieren (Kapitel 2.1.2). Statt dessen wird im Flüssigkeitsmodell die Last implizit transportiert. In jedem elementaren Lastverteilungsschritt verschieben stark belastete Prozessoren die Lastelemente an ihre direkten Nachbarn *(Shifts)*. Sie selbst erhalten wiederum unter Umständen Lasteinheiten von ihren Vorgängern. Andererseits geben schwach belastete Prozessoren keine Arbeit ab, sondern erhalten Lasteinheiten von ihren Vorgängern. Durch sukzessives Verschieben der Lastelemente wird die Last automatisch zu den schwach belasteten Prozessoren transferiert. Durch den hier verfolgten Ansatz der lokalen Lastverteilung sollen beide Ziele (Load-

sharing und Load-balancing) zwar nicht in einem Lastverteilungsschritt, aber dennoch asymptotisch mit ausreichend vielen Lastverteilungsschritten erreicht werden können.

4.1.2 Formales Modell

Für eine präzise Beschreibung der Lastverteilung nach dem Flüssigkeitsmodell und zur Formulierung von analytischen Aussagen über den Algorithmus (Kapitel 4.2) wird nun die oben dargestellte Idee formalisiert. Dabei wird als Verbindungstopologie der Prozessoren eine allgemeine zyklische Gitterstruktur angenommen:

Es sei ein D-dimensionaler, symmetrischer[1] Torus mit $P = K^D$ Prozessoren, für ein festes $K \in \mathbb{N}$, gegeben. Der Prozessor P_i hat eine eindeutige Identifikation i, die sich aus den (zyklischen) Koordinaten im Torus ergibt und durch den D-dimensionalen Vektor $i = (i_1, \ldots, i_D)$ bestimmt wird. Die Menge aller zulässigen Identifikatoren für die Prozessoren ist durch die Indexmenge gegeben:

$$I := \{(i_1, \ldots, i_D) \mid i_j \in \{1, \ldots, K\} \text{ und } j = 1, \ldots, D\} \tag{4.1}$$

Zur Vereinfachung werden diese Indizes (mod D) + 1 gerechnet, d.h. P_i steht für $P_{(i \bmod D)+1}$. Der Zugriff auf genau eine Dimension d erfolgt durch den Vektor 1_d, der als d-te Komponente eine 1 und ansonsten eine 0 stehen hat.

Das System aus Prozessoren wird nun ausschließlich zu diskreten Zeitpunkten $t \in \mathbb{N}$ betrachtet. Dabei handelt es sich um eine Reihe von aufeinanderfolgenden Systemzuständen. Zu jedem beliebigen Zeitpunkt $t \in \mathbb{N}$ sei die Belastung des Prozessors P_i durch die quantisierte Last $L_i(t) \in \mathbb{N}$ gegeben. Zur Vereinfachung können im folgenden eindeutige Zeitparameter entfallen, und die Lastzustände beziehen sich auf den Zeitpunkt t, d.h. L_i steht für $L_i(t)$.

Die Veränderung der Lastzustände wird durch Verschieben *(Shifts)* elementarer Lasteinheiten zwischen benachbarten Prozessoren bewerkstelligt. Die boolesche Funktion $C_{i,d}(t)$ steuert das Verschieben der Lasteinheiten. Falls für den Prozessor i die Bedingung $C_{i,d}(t)$ zutrifft, dann verschiebt er ein Lastelement in Richtung der Dimension d. Dabei steht $C_{i,d}(t)$ stellvertretend für eine der sechs Bedingungen C0 bis C5 aus Tabelle 4.1. Sie wertet die gewünschte Bedingung für den Prozessor i bzw. dessen Lastzustand L_i zum Zeitpunkt t aus. Dabei wird die eindimensionale Bedingung in der Dimension d angewendet. Das heißt, die skalaren Operationen der Bedingung beziehen sich ausschließlich auf die d-te Komponente des D-dimensionalen Vektors. Zum Beispiel steht die letzte Bedingung $C5_{i,d}(t)$ für $L_i > 0 \wedge L_{(i_1, \ldots, i_D)} \geq L_{(i_1, \ldots, i_d+1, \ldots, i_D)}$.

---

[1] Ein *symmetrischer* Torus hat in jeder Dimension dieselbe Anzahl von Elementen (Prozessoren). Die Symmetrie ist keine notwendige Voraussetzung für das Flüssigkeitsmodell, vereinfacht aber seine Darstellung.

	$C_{i,d}$ $\Leftrightarrow$	"Prozessor P_i verschiebt ein Lastelement an P_{i+1_d}" $\Leftrightarrow$
C0:	$L_i > 0$	"P_i hat ein Lastelement zum Transferieren"
C1:	$L_i > 1$	"P_i ist nach Abgabe eines Lastelements nicht unbeschäftigt"
C2:	$C1 \vee [(L_i = 1) \wedge (L_{i-1_d} > 1)]$	"P_i ist nach Abgabe eines Lastelements nicht unbeschäftigt oder bekommt ein Lastelement von P_{i-1_d} transferiert"
C3:	$C1 \wedge L_i \geq L_{i+1_d}$	"P_i ist nach Abgabe eines Lastelements nicht unbeschäftigt und P_{i+1_d} ist nicht höher belastet"
C4:	$C2 \wedge L_i \geq L_{i+1_d}$	"P_i ist nach Abgabe eines Lastelements nicht unbeschäftigt oder bekommt ein Lastelement von P_{i-1_d} transferiert und P_{i+1_d} ist nicht höher belastet"
C5:	$C0 \wedge L_i \geq L_{i+1_d}$	"P_i hat ein Lastelement zum Transferieren und P_{i+1_d} ist nicht höher belastet"

Tabelle 4.1: Formale und verbale Beschreibung von verschiedenen Instanziierungen (C0 bis C5) der Shift-Bedingung $C_{i,d}$*, welche abhängig von der Last* L_i *angibt, ob der Prozessor* i *ein Lastelement an seinen Nachbarn in der Dimension* d *transferieren soll*

Die Bedingungen C0 und C1 verwenden nur Lastinformation L_i des Prozessors *i*. Die Bedingung C1 und ihre Erweiterung in C2 garantiert, daß keiner der belasteten Prozessoren durch Verschieben von Lastelementen unbeschäftigt wird. Die Bedingungen C2 bis C5 benutzen zusätzlich Lastinformation des Vorgängers L_{i-1_d} oder des nachfolgenden Prozessors L_{i+1_d}. Durch jede einzelne dieser Bedingungen wird ein eigener Mechanismus zur Lastverteilung im Rahmen des Flüssigkeitsmodells festgelegt.

Als ein Lastverteilungsschritt des Flüssigkeitsmodells wird der Zustandsübergang von dem Zeitpunkt t nach $t + 1$ betrachtet. Ein solcher Lastverteilungsschritt besteht wiederum aus mehreren Teilschritten. Dazu wird in jeder Dimension $d = 1, \ldots, D$ der Lastzustand L_i aller Prozessoren $i = 1, \ldots, P$ synchron verändert. Statt nun auf den expliziten Transfer von Last einzugehen, wird hier nur der resultierende Effekt betrachtet. Die Zustandsveränderung der Last hängt nur von den direkten Nachbarn innerhalb einer Dimension ab. Damit kann eine Definition der Lastverteilung abhängig von den Shift-Bedingungen angegeben werden.

Prozessor:	P_{i-1_d}		P_i	
Shift-Bedingung:	$C_{i,d}(t)$ = False		$C_{i,d}(t)$ = False	
Shift eines Lastelements:		–		–
Zustandsänderung:			$L_i(t+1) := L_i(t)$	
Shift-Bedingung:	$C_{i,d}(t)$ = False		$C_{i,d}(t)$ = True	
Shift eines Lastelements:		–		• →
Zustandsänderung:			$L_i(t+1) := L_i(t) - 1$	
Shift-Bedingung:	$C_{i,d}(t)$ = True		$C_{i,d}(t)$ = False	
Shift eines Lastelements:		• →		–
Zustandsänderung:			$L_i(t+1) := L_i(t) + 1$	
Shift-Bedingung:	$C_{i,d}(t)$ = True		$C_{i,d}(t)$ = True	
Shift eines Lastelements:		• →		• →
Zustandsänderung:			$L_i(t+1) := L_i(t)$	

Tabelle 4.2: Lastverteilung nach dem Flüssigkeitsmodell für Prozessor P_i mit seinem Vorgänger P_{i-1_d}: Auf der booleschen Auswertung der Shift-Bedingungen $C_{i,d}$ basieren die Shifts der Lastelemente, welche ihrerseits die Zustandsänderung in der Prozessorbelastung von P_i ergeben

Definition 4.1: (Flüssigkeitsmodell)
Eine Zustandsveränderung $L_i(t) \rightarrow L_i(t+1)$, mit $i \in I$, heißt Lastverteilung nach dem Flüssigkeitsmodell, *oder engl.* Liquid Model (LM-C, *mit Bedingung C aus Tabelle 4.1), genau dann, wenn sukzessive in jeder Dimension $d = 1, \ldots, D$ folgende Gleichung angewendet wird:*

$$L_i(t+1) := \begin{cases} L_i(t) + 1, & \text{falls } C_{i-1_d,d}(t) \wedge \neg C_{i,d}(t) \\ L_i(t) - 1, & \text{falls } \neg C_{i-1_d,d}(t) \wedge C_{i,d}(t) \\ L_i(t), & \text{sonst} \end{cases} \quad \Diamond$$

In Tabelle 4.2 ist der Mechanismus des Lasttransfers abhängig von den Shift-Bedingungen veranschaulicht. Für jede mögliche Auswertung der Shift-Bedingungen zweier direkt benachbarter Prozessoren sind die daraus resultierenden Lasttransfers angegeben. Das Verschieben von Lastelementen hat wiederum eine Veränderung der Lastzustände der Prozessoren zur Folge. Diese Veränderung der Lastzustände dient in Definition 4.1 als Grundlage zur Definition des Flüssigkeitsmodells.

Mit der obigen Definition ist der elementare Schritt der iterativen Lastverteilung nach dem Flüssigkeitsmodell festgelegt. Nun stellt sich die Frage nach dem Ziel und der damit verbundenen Terminierung der Lastverteilung. Das Ziel einer Lastverteilungsmethode beim Load-balancing ist es, von jeder beliebigen Verteilung der Last L_i mit möglichst wenigen Zustandsübergängen $t \rightarrow t + 1$ eine ausgeglichene Lastkonfiguration zu erreichen. Dabei soll die Last

der Prozessoren ungefähr der mittleren Last entsprechen. Dieser ausgeglichene Lastzustand kann über die Abweichung von der mittleren Last wie folgt definiert werden.

Definition 4.2: (Balanciertheit)
Eine Lastkonfiguration L_i, mit $i \in I$ heißt balanciert *oder* ausgeglichen *genau dann, wenn für alle $i \in I$ gilt:*

$$|L_i - \bar{L}| \leq 1, \text{ mit der mittleren Last } \bar{L} = \frac{1}{P}\sum_{j \in I} L_j \qquad \Diamond$$

Die obige Definition dient als formales Abbruchkriterium der Lastverteilung. Solange die Bedingung für einen ausgeglichenen Lastzustand nicht erfüllt wird, ist die sukzessive Anwendung von einzelnen Lastverteilungsschritten notwendig. Als Operation zum Verschieben der Last wird eine Prozedur Transfer_Load(P_i, P_{i+1_d}) angenommen, die eine Lasteinheit von Prozessor P_i zu seinem Nachbarn P_{i+1_d} transportiert. Damit ergibt sich eine imperative Schreibweise der Lastverteilung aus Definition 4.1:

Algorithmus 4.1: (Flüssigkeitsmodell)

```
while (|L_i - L̄| > 1) do
    for d = 1, ..., D do
        for all processors P_i, i ∈ I, do in parallel
            if C_{i,d} then
                Transfer_Load(P_i, P_{i+1_d});
                /* L_i := L_i - 1 and L_{i+1_d} := L_{i+1_d} + 1 */
        end;
    end;
end;
```

Bei der Integration der Lastverteilung nach dem Flüssigkeitsmodell in einen konkreten Anwendungsalgorithmus wird das formale Abbruchkriterium nicht abgeprüft. Dazu wäre globale Information über den Systemzustand notwendig, deren Berechnung die Skalierbarkeit des Algorithmus beeinträchtigt. Ohne ein Abbruchkriterium kann die Lastverteilung von sich aus nicht terminieren. Dies ist auch nicht notwendig, da nach dem Ablaufschema aus Kapitel 1.6 die Lastverteilung und die Anwendung nebenläufig ausgeführt werden. Das heißt, durch einen periodischen Trigger-Mechanismus (Kapitel 4.4) werden alternierend jeweils einige Iterationen der beiden Prozesse initiiert. Die Terminierung der Lastverteilung wird also durch die Terminierung der Anwendung garantiert.

4.1.3 Beispiel

Das Beispiel in Abb. 4.2 zeigt einen einzelnen Lastverteilungsschritt des Flüssigkeitsmodells in einem Ring. Dabei gibt ein Prozessor genau dann

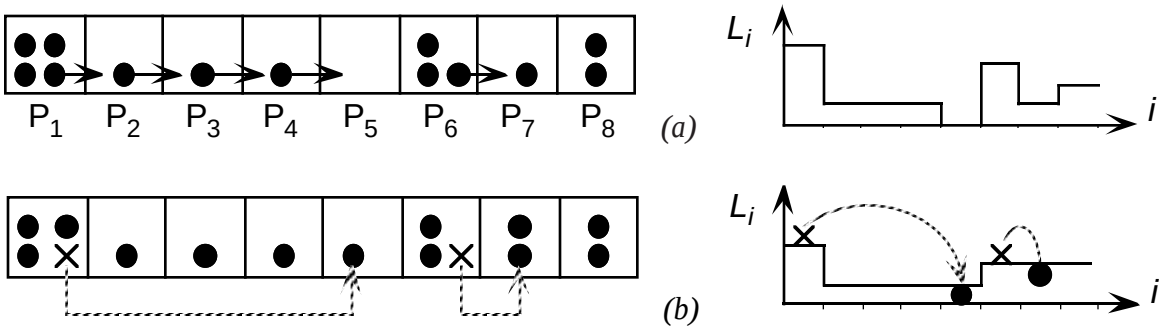

Abb. 4.2: Beispiel für einen einzelnen Lastverteilungsschritt des Flüssigkeitsmodells mit Bedingung C5 in einem Ring von acht Prozessoren P_1 bis P_8 (schwarze Pfeile: realer Lasttransfer, graue Pfeile: virtueller Lasttransfer). Die Lastkonfigurationen sind links als Prozessoren mit Lastelementen und rechts als Funktionsgraph der Belastung L_i über die Prozessoren P_i dargestellt.

Lastelemente an seinen Nachbarn, wenn Bedingung C5 zutrifft – also, wenn ein Prozessor überhaupt Lastelemente besitzt und seine Last größer oder gleich als die seines Nachfolger ist. Außerdem verschieben laut Definition 4.1 die Prozessoren die Lastelemente synchron in die gleiche Richtung, welche durch die Indizierung vorgegeben ist (hier: nach rechts). In (a) ist die Ausgangssituation gegeben und die Shifts sind durch Pfeile angedeutet. In (b) ist die Situation nach der Shift-Operation gegeben. Die Pfeile geben die "virtuellen" Transfer von Lastelementen an.

Zwei Effekte des Flüssigkeitsmodells können an diesem Beispiel gesehen werden. In der linken Prozessorgruppe (P_1 bis P_5) wird ein Lastelement von P_1 nach P_5 virtuell, ausschließlich durch lokale Operationen, transferiert. In der rechten Gruppe (P_6 bis P_8) findet ein Lastausgleich zwischen den Prozessoren statt.

Wird das Flüssigkeitsmodell in Kombination mit dem B&B-Algorithmus eingesetzt, dann entsprechen die Lastelemente den noch zu bearbeitenden Knoten bzw. Teilproblemen in den lokalen OPEN-Mengen. Für jede Shift-Operation wird ein Knoten von OPEN ausgewählt und ein erhaltener Knoten wieder eingefügt. Falls bei jedem Shift immer die nach der heuristischen Bewertung "besten" Knoten ausgewählt werden, dann wird eine globale Bestensuche angenähert. Dies wird für kordale Ringe in [Wah84b] gezeigt.

Ein Vergleich des Flüssigkeitsmodells mit der bekannten Mittelungsmethode NNA aus Kapitel 2.2.3 zeigt zwei wichtige Unterschiede. Zum einen sind in NNA keine dieser globalen Effekte durch ausschließlich lokale Shift-Operationen möglich. Dies ist darin begründet, daß zwischen zwei gleichbelasteten Prozessoren im NNA keine Lastelemente transferiert werden. Dieses balancierte Prozessorpaar bildet eine Hürde für den (virtuellen) Lasttransfer. Zum anderen erreicht NNA Load-sharing nur als Nebeneffekt von Load-balancing. Dies steht im Gegensatz zum Flüssigkeitsmodell. Dort wird in

erster Linie die Last zwischen den Prozessoren aufgeteilt (Load-sharing), und erst in zweiter Linie findet ein Ausgleich (Load-balancing) statt. Im folgenden Kapitel wird dieser Effekt detaillierter besprochen.

4.2 Analytische Betrachtungen

In diesem Kapitel wird der formale Rahmen aus dem letzten Kapitel verwendet, um drei grundlegende Aussagen über das Flüssigkeitsmodell zu treffen: (1) die globalen Effekte durch lokale Operationen, (2) die Konvergenz zu einem balancierten Zustand und (3) die Effizienz des Algorithmus. Alle Aussagen werden im folgenden analytisch hergeleitet. Dabei kann der Lastverteilungsprozeß für sich alleine betrachtet werden. Da sich in Kapitel 4.3 die Ausprägung des Flüssigkeitsmodells mit Shift-Bedingung C5 als die effizienteste erweist, beziehen sich die Aussagen der Analyse ausschließlich auf diese Bedingung. Für die anderen Ausprägungen können ähnliche Aussagen abgeleitet werden.

Ein Vertreter von globalen Effekten wurde schon an einem Beispiel mit LM-C5 in Abb. 4.2 illustriert[2]. Der virtuelle Lasttransfer findet in Richtung der Shifts statt (nach rechts). Voraussetzung dafür ist eine Reihe mit Prozessoren, die mit genau einer Lasteinheit belastet sind und einige unbelastete Prozessoren rechts davon. Ein weiterer, allgemeiner Vertreter der globalen Effekte stellt der virtuelle Lasttransfer entgegen der Shift-Richtung dar. Dabei wird die Last um eine Einheit bei einem Prozessor verringert und bei einem anderen Prozessor, mit kleinerem Index, erhöht. Die Belastung der dazwischenliegenden Prozessoren bleibt unverändert. Dieser Effekt wird in folgender Definition präzisiert (Beispiele in Abb. 4.3).

Definition 4.3: (virtueller Lasttransfer)
Seien zwei verschiedene Prozessoren P_x und P_y, innerhalb einer Dimension d gegeben, d.h. es existiert ein $k \in \mathbb{N}^+$ mit $y = x + k \cdot 1_d$. Eine Veränderung einer gegebenen Lastkonfiguration $L_i(t)$, mit $i \in I$, heißt virtueller Lasttransfer *zwischen den Prozessoren P_x und P_y, falls gilt: $L_x(t+1) = L_x(t) + 1$ und $L_y(t+1) = L_y(t) - 1$ unter der Bedingung, daß für alle $i = x+1_d, \ldots, y-1_d$ gilt $L_i(t+1) = L_i(t)$.* ◊

Neben den globalen Effekten durch die Lastverteilungsmethode können in der Verteilung der Last selbst globale Strukturen beobachtet werden. Da die Lastverteilung in jeweils nur einer Dimension arbeitet, sind auch nur eindimensionale Strukturen von Interesse. Eine typische Struktur stellt der streng monotone Anstieg der Prozessorbelastung in Shift-Richtung dar. Solche

---

[2] Ein ausführlicheres Beispiel ist in Tabelle 4.3 angegeben.

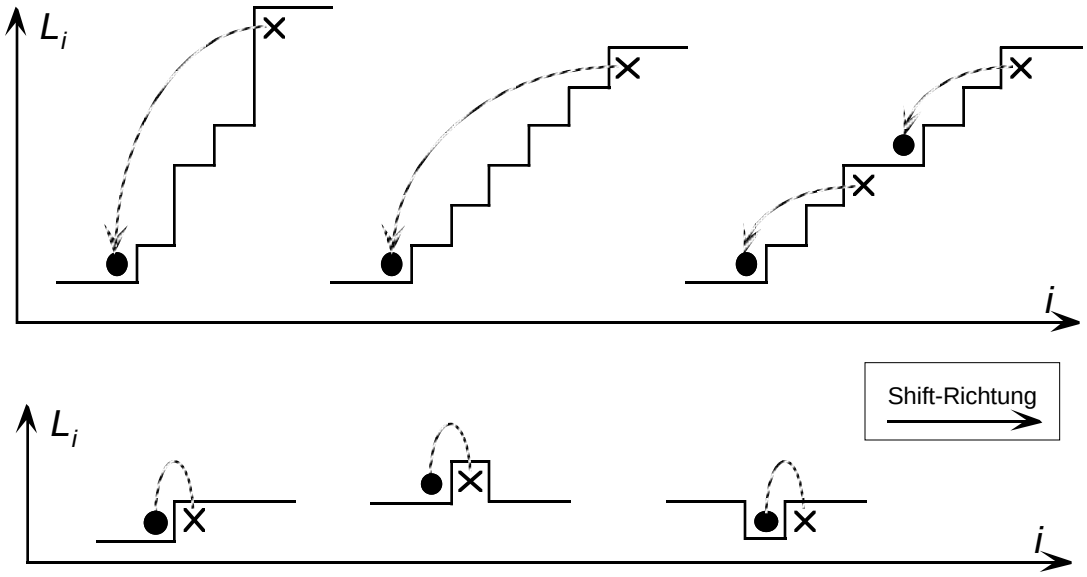

Abb. 4.3: Verschiedene Formen von Rampen in eindimensionalen Lastkonfigurationen und den daraus resultierenden virtuellen Lasttransfers durch das Flüssigkeitsmodell LM-C5 im eindimensionalen Ring

Anstiege der Belastung mit maximaler Länge werden in folgender Definition als Rampen bezeichnet (siehe auch die Beispiele in Abb. 4.3).

Definition 4.4: (Rampen)
Die Lastkonfiguration zwischen zwei Prozessoren P_x und P_y, innerhalb einer Dimension d bildet eine Rampe, *wenn folgende vier Bedingungen zutreffen:*

(1) *es existiert ein $k \in \mathbb{N}^+$ mit $y = x + k \cdot 1_d$ und*
(2) *für alle $i = x, x+1_d, \ldots, y-1_d$ gilt $L_i < L_{i+1_d}$ und*
(3) *$L_{x-1_d} \geq L_x$ und*
(4) *$L_{y+1_d} \leq L_y$* ◊

In der Abb. 4.3 ist eine Reihe von Beispielen für Rampen in eindimensionalen Lastkonfigurationen dargestellt. Die Belastung der Prozessoren L_i ist gegenüber dem Prozessorindex i aufgetragen. Die Shifts erfolgen nach rechts. In der oberen Reihe nimmt die Steigung der Rampen von links nach rechts ab. In der unteren Reihe sind Spezialfälle für die kleinst-möglichen Rampen in Belastungsplateaus dargestellt (links: Sprung, Mitte: Maximum, rechts: Minimum). Zusätzlich sind in dieser Abbildung auch die virtuellen Lasttransfers eingetragen. Der Zusammenhang zwischen den Rampen und den virtuellen Lasttransfers wird in dem folgenden Satz hergestellt.

Satz 4.5: (globale Effekte)
Sei eine Lastkonfiguration $L_i > 0$, mit $i \in I$, gegeben.[3] Ein virtueller Lasttransfer zwischen den Prozessoren P_x und P_y findet bei einem Lastverteilungsschritt nach LM-C5 statt, wenn die Lastkonfiguration zwischen diesen beiden Prozessoren eine Rampe bildet.

Beweis. Die Bedingung (1) aus Definition 4.4 garantiert die in Definition 4.3 geforderte Reihenfolge der Prozessoren P_x und P_y. Aus den Bedingungen (3) und (2) und der Gleichung in Definition 4.2 ergibt sich, daß P_x ein zusätzliches Lastelement von P_{x-1_d} erhält aber keines an P_{x+1_d} abgibt, d.h. nach einem Lastverteilungsschritt hat der Prozessor P_x ein Lastelement mehr bzw. $L_x(t+1) = L_x(t) + 1$. Aus den Bedingungen (4) und (2) ergibt sich analog, daß P_y ein Lastelement an P_{y+1_d} abgibt und keines von P_{y-1_d} erhält, d.h. nach einem Lastverteilungsschritt hat der Prozessor P_y ein Lastelement weniger bzw. $L_y(t+1) = L_y(t) - 1$. Für die Prozessoren P_j mit $j = x+1_d, \ldots, y-1_d$ folgt aus Bedingung (2) mit $i = j$, daß P_j keine Last abgibt, und aus Bedingung (2) mit $i = j-1$, daß P_j keine Last erhält. Damit sind die Lastzustände der Prozessoren P_j nach dem Lastverteilungsschritt unverändert bzw. $L_j(t+1) = L_j(t)$. Nun sind die Anforderungen aus Definition 4.3 erfüllt. ◊

Eine Grundvoraussetzung für die Konvergenz der Lastverteilungsmethode ist, daß die Qualität der Lastkonfiguration durch Anwendung eines Lastverteilungsschritts nicht verschlechtert wird. Mit dem LM-C5 kann dieses konservative Verhalten garantiert werden. Der Grund dafür ist, daß das Maximum bzw. das Minimum der Last nicht erhöht bzw. erniedrigt wird. In folgendem Satz wird diese Aussage gezeigt.

Satz 4.6: (Konservativität)
Nach Anwendung eines Lastverteilungsschritts durch LM-C5 auf eine Lastkonfiguration $L_i(t)$, mit $i \in I$ und $L_{max}(t) = max\{L_i(t) \mid i \in I\}$ bzw. $L_{min}(t) = min\{L_i(t) \mid i \in I\}$, gilt:

$$L_{max}(t+1) \leq L_{max}(t) \text{ und } L_{min}(t+1) \geq L_{min}(t).$$

Beweis. Zunächst wird nur der Teilschritt in Dimension d betrachtet. Durch die letzten beiden Fälle der Zustandsveränderung nach dem LM aus Definition 4.1 kann der Lastzustand eines Prozessors nicht weiter erhöht werden. Die erste Zustandsänderung lautet $L_i(t+1) := L_i(t) + 1$ und wird durchgeführt, falls $C_{i,d}(t) \wedge \neg C_{i+1_d,d}(t)$ gilt. Mit C5 als Bedingung ist dies äquivalent zu $L_{i-1_d}(t) \geq L_i(t) \wedge L_i(t) < L_{i+1_d}(t)$. Damit hat nach einem Lastverteilungs-Teilschritt in der

[3] In diesem und den folgenden Sätzen wird von einer Lastkonfiguration ausgegangen, in der alle Prozessoren beschäftigt sind ($L_i > 0$, mit $i \in I$), d.h. das Ziel Load-sharing ist schon erreicht. Diese Annahme ist unkritisch, da von dem Flüssigkeitsmodell alle Prozessoren in kurzer Zeit ($O(D \cdot K)$ Schritten) mit Lastelementen versorgt werden.

Dimension d der Prozessor P_i höchstens gleich viel Last wie sein Nachbar P_{i+1_d}. Für die minimale Last ist die Argumentation analog. Da die Teilschritte in den verschiedenen Dimensionen nacheinander abgearbeitet werden, erhöht bzw. erniedrigt auch ihre Kombination in einem Lastverteilungsschritt die Maxima bzw. Minima nicht. ◊

Neben der Konservativität bezüglich der Lastkonfiguration, muß die Lastverteilungsmethode zusätzlich die Verteilung der Last verbessern bzw. ausgleichen. Bei iterativen Verfahren erfordert dies innerhalb einer festen Anzahl von Iteration einen, wenn auch noch so kleinen, Fortschritt – andernfalls ist die Konvergenz des Verfahrens nicht garantiert. In dem LM-C5 ist der Fortschritt an allen linken Maximumstellen bzw. rechten Minimumstellen in Hochplateaus bzw. Tiefebenen der Lastverteilung festzustellen. An diesen Stellen ist das Verhalten symmetrisch, so daß hier nur eine linke, globale Maximumstelle betrachtet wird. Die Anwendung eines Lastverteilungsschritt kann zwei alternative Auswirkungen haben. Entweder wird die Maximumstelle in ihrem Wert um eine Lasteinheit vermindert (siehe obere Reihe in Abb. 4.3). Da keine anderen Extremstellen spontan auftreten können, hat sich damit die Gesamtzahl von Maximumstellen reduziert. Oder die Maximumstelle ist entgegen der Shift-Richtung um einem Prozessor nach links gewandert (siehe untere Reihe in Abb. 4.3). Da symmetrisch dazu auch eine Minimumstelle nach rechts wandert und sich beide Extrema irgendwann ausgleichen werden, stellt dies auch einen Fortschritt dar. In folgendem Satz werden beide Alternativen des Fortschritts gezeigt.

Satz 4.7: (Konvergenz)
Sei eine nicht-ausgeglichene Lastkonfiguration $L_i > 0$, *mit* $i \in I$ *und* $L_{max} = max\{L_i \mid i \in I\}$, *gegeben. Nach einem Lastverteilungsschritt durch LM-C5 ist jede Maximumstelle* $m \in I$ *am linken Ende eines Hochplateaus* ($L_m(t) = L_{max}(t)$ *und* $L_{m-1_d}(t) < L_{max}(t)$) *entweder:*

(1) in ihrem Wert um eine Lasteinheit reduziert:
$L_m(t+1) = L_{max}(t) - 1$ *und* $L_{m-1_d}(t+1) < L_{max}(t)$ *oder*

(2) um ein Prozessor nach links gewandert:
$L_m(t+1) = L_{max}(t) - 1$ *und* $L_{m-1_d}(t+1) = L_{max}(t)$

Beweis. Da die Lastkonfiguration $L_i(t)$ nicht ausgeglichen ist, existieren immer eine Dimension d und ein Index $m \in I$ mit $L_m = L_{\max}$ und $L_{m-1_d} < L_m$. Damit bildet P_m das obere (rechte) Ende einer Rampe aus Definition 4.4. Sei P_n, mit $n < m$, das dazugehörige untere (linke) Ende. Damit gibt $L_m - L_n$ die Lastdifferenz der Rampe an.

Fall 1. Falls $L_{m-2\cdot 1_d} < L_{m-1_d}$ oder $L_{m-1_d} + 1 < L_m$ gilt, dann ist die Lastdifferenz echt größer Eins. Laut Satz 4.5 wird nach dem Lastverteilungsschritt ein Lastelement von P_m virtuell nach P_n transferiert. Da $L_m(t+1) = L_m(t) - 1$ gilt, ist P_m keine

Maximumstelle der bisherigen Größe mehr, d.h. $L_m(t+1) < L_{\max}(t)$. Da die Lastdifferenz echt größer Eins ist, entsteht auch keine andere globale Maximumstelle am unteren Rampenende und Bedingung (1) ist erfüllt.
Fall 2. Nun gelte $L_{m-2\cdot 1_d} \geq L_{m-1_d}$ und $L_{m-1_d} + 1 = L_m$. Damit bildet der direkt benachbarte Prozessor P_{m-1_d} das linke Rampenende, und die Lastdifferenz der Rampe ist genau Eins. Laut Satz 4.5 wird die Maximumstelle von P_m virtuell nach P_{m-1_d} transferiert, mit $L_m(t+1) = L_m(t) - 1$. Durch die kleine Lastdifferenz bildet P_{m-1_d} nach dem Lasttransfer eine neue globale Maximumstelle, d.h. $L_{m-1_d}(t+1) = L_{\max}(t)$. Die Maximumstelle von P_m ist damit nach links zu P_{m-1_d} gewandert und Bedingung (2) ist erfüllt. ◊

Beide der oben gezeigten Möglichkeiten des Fortschritts tragen zu einem stückweisen Ausgleich der Last und damit zur Konvergenz des Verfahrens bei. Daraus läßt sich folgende obere Abschätzung für den Zeitaufwand des LM-C5 aufstellen:

Satz 4.8: (Zeitaufwand)
Um eine unbalancierte Lastkonfiguration $L_i > 0$, mit $i \in I$ und $L_{diff} = max\{ |L_i - L_j| , i, j \in I\}$, in einem symmetrischen, D-dimensionalen Torus mit $P = K^D$ Prozessoren, $K \in \mathbb{N}$, durch LM-C5 auszugleichen ist ein maximaler Zeitaufwand T (gemessen in Lastverteilungsschritten) notwendig von:

$$T = O(D \cdot K \cdot L_{diff})$$

Beweis. Es reicht zu zeigen, daß nach $O(D \cdot K)$ Lastverteilungsschritten das globale Maximum um mindestens eine Lasteinheit reduziert wird, d.h. $L_{max}(t + O(D \cdot K)) < L_{max}(t)$ gilt. Dazu wird die Menge von globalen Maximumstellen betrachtet. Sei $m \in I$ mit $L_m = L_{max}$ eine solche Stelle.
Fall 1. Sei $L_{m-1_d} < L_{max} - 1$. Laut Satz 4.7 wird durch einen Lastverteilungsschritt die Maximumstelle m abgebaut.
Fall 2. Sei $L_{m-1_d} \leq L_{max} - 1$. Laut Satz 4.7 wird durch einen Lastverteilungsschritt die Maximumstelle in m innerhalb einer Dimension nach links auf $m - 1_d$ verschoben. Da die Lastkonfiguration unbalanciert ist, existiert eine dazugehörende Minimumstelle, welche ihrerseits nach rechts wandert. Bis sich beide treffen bedarf es höchstens $K / 2$ Schritte. Liegen die korrespondierenden Extremstellen in unterschiedlichen Dimensionen, so sind höchstens $D \cdot K / 2$ Schritte notwendig.
Fall 3. Sei $L_{m-1_d} = L_{max}$. Die Stelle m wird erst dann verändert, wenn alle direkt links von m liegenden Maximumstellen verschoben wurden und $L_{m-1_d} < L_{max}$ gilt. Im schlimmsten Fall müssen dazu bis zu $P - 2$ Stellen verändert werden. Dann trifft der Fall 2 zu.
In allen Fällen sind höchstens $O(D \cdot K)$ Lastverteilungsschritte notwendig, bis die Maximumstelle m reduziert wird. Da diese Stelle stellvertretend für alle Maximumstellen steht und alle Stellen parallel bearbeitet werden, gilt

$L_{max}(t + O(D \cdot K)) < L_{max}(t)$. Mit der maximalen Lastdifferenz L_{diff} folgt dann direkt der Satz. ◊

Dieser letzte Satz zeigt, daß eine lokale Lastverteilung sehr wohl effizient sein kann. Gegenüber dem NNA, aus Kapitel 2.2.3, weist das Flüssigkeitsmodell auf SIMD-Rechnern mit dem Torus als Verbindungsnetzwerk nur einen linearen Aufwand in den einzelnen Komponenten auf.[4] Der NNA-Algorithums als Spezialfall der Diffusionsansätze benötigt hingegen einen quadratischen Aufwand abhängig von der maximalen Prozessoranzahl pro Dimension.

Bei dem Aufwandsvergleich von LM-C5 und NNA ist zu beachten, daß zwei unterschiedliche Maße angewendet werden. Der lineare Aufwand von LM-C5 ist gemessen in der kommunizierten Datenmenge. Der quadratische Aufwand von NNA ist gemessen in der Anzahl von Datentransfers. Die Menge ist immer größer oder gleich der Anzahl von Datentransfers, da für die Kommunikation von einer Informationseinheit mindestens ein Aufsetzen der Verbindung notwendig ist. Daher ist der Vergleich der unterschiedlichen Aufwandsmaße gerechtfertigt.

4.3 Simulation des Lastverteilungsprozesses

Der B&B-Algorithmus mit dynamischer Lastverteilung kann als zwei verwobene, gegeneinander spielende Prozesse angesehen werden (Kapitel 1.5). Der B&B-Prozeß verschlechtert die Verteilung der Last durch Abarbeiten des Suchbaums. Dabei werden durch Expansionen neue Knoten in die OPEN-Menge hinzugefügt oder durch heuristische Beschneidungen existierende Knoten beschnitten. In beiden Fällen wird die Last der Prozessoren in nicht vorherzusehender Weise verändert. Der Lastverteilungsprozeß versucht durch Verschieben der Knoten die Last wieder auszugleichen, so daß jeder Prozessor ungefähr gleich viel Last enthält. Betrachtet man nur den Lastverteilungsprozeß unabhängig von dem B&B-Prozeß, dann ist seine Wirkungsweise deutlicher zu erkennen.

In einer Simulation werden NNA und LM für verschiedene Shift-Bedingungen im Ring mit P Prozessoren verglichen. Als ungünstigen Fall wird angenommen, daß der erste Prozessor die gesamte Last $L_{sum} = c \cdot P$ mit $c > 0$ als ganzzahlige Konstante enthält, und die restlichen Prozessoren unbeschäftigt sind. In dem ausgeglichenen Systemzustand ist jeder Prozessor mit c Einheiten belastet. Jede Lastverteilungsmethode wird ausgeführt, bis der balancierte Zustand erreicht ist. Für den NNA wird der Algorithmus aus Kapitel 2.2.3 verwendet, wobei nur die Differenz zweier entgegengesetzter Lasttransfers

[4] Für die gebräulichen Topologien wächst entweder nur die Anzahl von Prozessoren K pro Dimension oder die Dimension D selbst, aber nicht beide gemeinsam.

T:	LM-C5:								NNA:							
0	16	0	0	0	0	0	0	0	16	0	0	0	0	0	0	0
1	15	1	0	0	0	0	0	0								
2	14	1	1	0	0	0	0	0								
3	13	1	1	1	0	0	0	0								
4	12	1	1	1	1	0	0	0								
5	11	1	1	1	1	1	0	0								
6	10	1	1	1	1	1	1	0								
7	9	1	1	1	1	1	1	1								
8	8	1	1	1	1	1	1	2								
9	7	1	1	1	1	1	2	2								
10	6	1	1	1	1	2	1	3								
11	5	1	1	1	2	1	2	3	5	6	0	0	0	0	0	5
12	4	1	1	2	1	2	2	3								
13	3	1	2	1	2	1	3	3								
14	3	2	1	2	1	2	2	3	5	4	2	0	0	0	1	4
15	3	2	2	1	2	1	3	2								
16	2	2	2	2	1	2	2	3	4	4	2	1	0	0	2	3
17	2	2	2	2	2	1	3	2	4	3	2	1	1	0	2	3
18	2	2	2	2	2	2	2	2	3	3	2	2	0	1	2	3
19									3	2	3	1	1	1	2	3
20									2	3	2	2	1	1	2	3
21									3	2	3	1	2	1	2	2
22									2	3	2	2	1	2	2	2
23									2	2	3	1	2	2	2	2
24									2	2	2	2	2	2	2	2

Tabelle 4.3: Beispiel für die Bearbeitung des ungünstigsten Falls durch das Flüssigkeitsmodell (LM-C5) und die Nachbarschafts-Mittelung (NNA) mit acht Prozessoren in einem Ring über die Zeit T *aufgetragen*

ausgetauscht wird. Falls ein nicht-ganzzahliger Betrag von Last verschoben werden sollte, dann wird der Betrag asymmetrisch gerundet. Bei Transfers nach rechts wird aufgerundet und bei Transfers nach links abgerundet. Dies ermöglicht, daß der ausgeglichene Systemzustand tatsächlich erreicht wird und nicht nur eine abfallende "Rampe" entsteht.

Als Zeiteinheit *T* dient die Anzahl von synchron durchgeführten Shifts. Da NNA mehrere Shift pro logischen Lastverteilungsschritt (NNA-Iteration) benötigt, wird die maximale Anzahl von Lasteinheiten-Transfers pro einzelne NNA-Iteration aufsummiert. Für das LM ist dies genau ein Shift. Dieses Zeitmaß ist vor allem dann sinnvoll, wenn die Initialisierungszeiten der Kommunikation klein sind gegenüber dem eigentlichen Lasttransfer. Dieser Umstand trifft vor allem auf die hier betrachteten feinkörnigen SIMD-Rechner zu, da die Prozessoren eng-gekoppelt sind.

In Tabelle 4.3 ist ein Beispiel für die Abarbeitung des ungünstigsten Falls durch NNA und LM angegeben. In einem Ring mit acht Prozessoren hält zunächst nur ein Prozessor 16 Lasteinheiten. Für beide Methoden sind nur die Zustände zwischen den Iterationen eingetragen. Dabei ist zu beachten, daß NNA

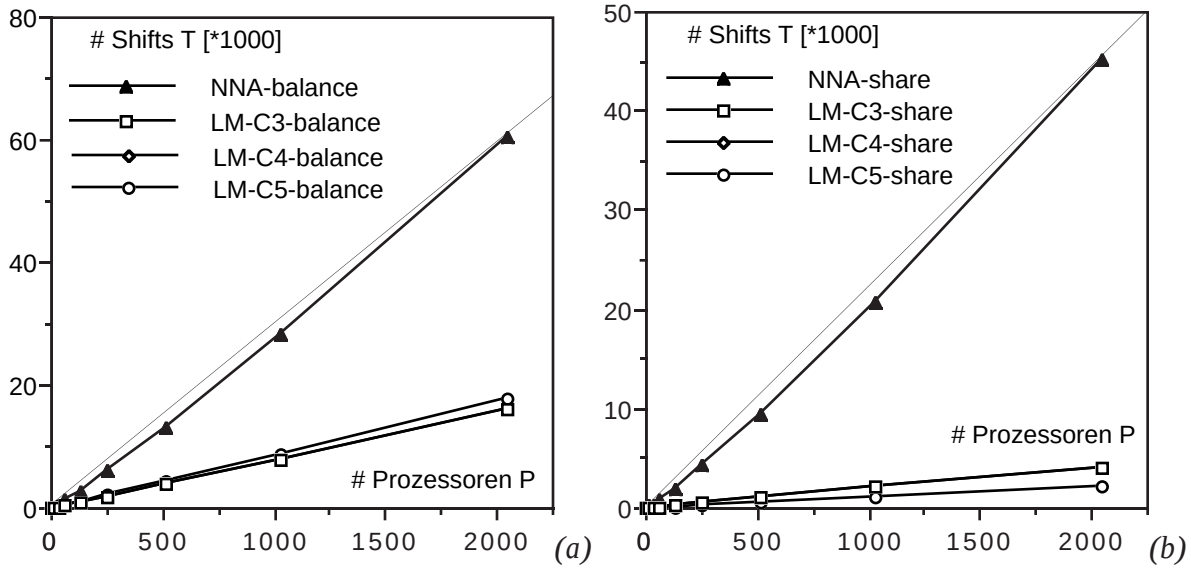

Abb. 4.4 Simulationsergebnisse für die Nachbarschafts-Mittelung (NNA) und dem Flüssigkeitsmodells (LM) bei synchrone Bearbeitung des ungünstigsten Fall (c = 5) im Ring bis das Ziel Load-balancing (a) bzw. Load-sharing (b) erreicht wurde

für eine Iteration mehrere Shifts benötigt. Noch bevor NNA die Mittelung zwischen den ersten beiden Prozessoren beenden konnte, hat das LM schon das Ziel Load-sharing nach dem siebten Shift erreicht. Nach dem 18. Shift hat LM auch das Load-balancing Ziel erreicht, ein Schritt bevor NNA ein Load-sharing erreicht. NNA benötigt hingegen noch weitere 12 Shifts für dieses Ziel. Insgesamt ist in diesem Beispiel LM um 25 % effizienter als NNA.

Die Allgemeingültigkeit des Verhältnisses der Laufzeitkonstanten aus dem obigen Beispiel zeigten sich in weiteren Simulationen. In Abb. 4.4 sind die Simulationsergebnisse für verschiedene Prozessoranzahlen angegeben. Die Ergebnisse zeigen einen fast nur linearen Anstieg der Zeit mit wachsendem P für alle betrachteten Lastverteilungsmethoden. Bei einer numerischen Anpassung von analytischen Kurven zeigen sich bei NNA quadratische Terme mit relativ kleinen Koeffizienten. Für Load-balancing benötigt NNA ungefähr viermal so lang wie LM. Für Load-sharing ist der Unterschied noch krasser. NNA ist 23 mal langsamer als LM.

Im LM werden die Lastelemente immer verschoben, außer die Lastdifferenz zu dem Nachfolger ist negativ (C3 bis C5). Daher hat Load-sharing gegenüber Load-balancing Priorität in LM. Im ungünstigsten Fall (siehe oben) wird nur dann die Last balanciert, wenn alle Prozessoren mit Arbeit versorgt sind. Der Zeitbedarf von LM für Load-sharing beträgt $T = P - 1$, gemessen in benötigter Shifts. Dies erklärt die sehr große Differenz zwischen LM und NNA beim Load-balancing.

In Abb. 4.5 ist das Laufzeitverhalten von NNA genauer untersucht. Abhängig von der initialen Last $L_{sum} = c \cdot P$ des ersten Prozessor, ist die normierte

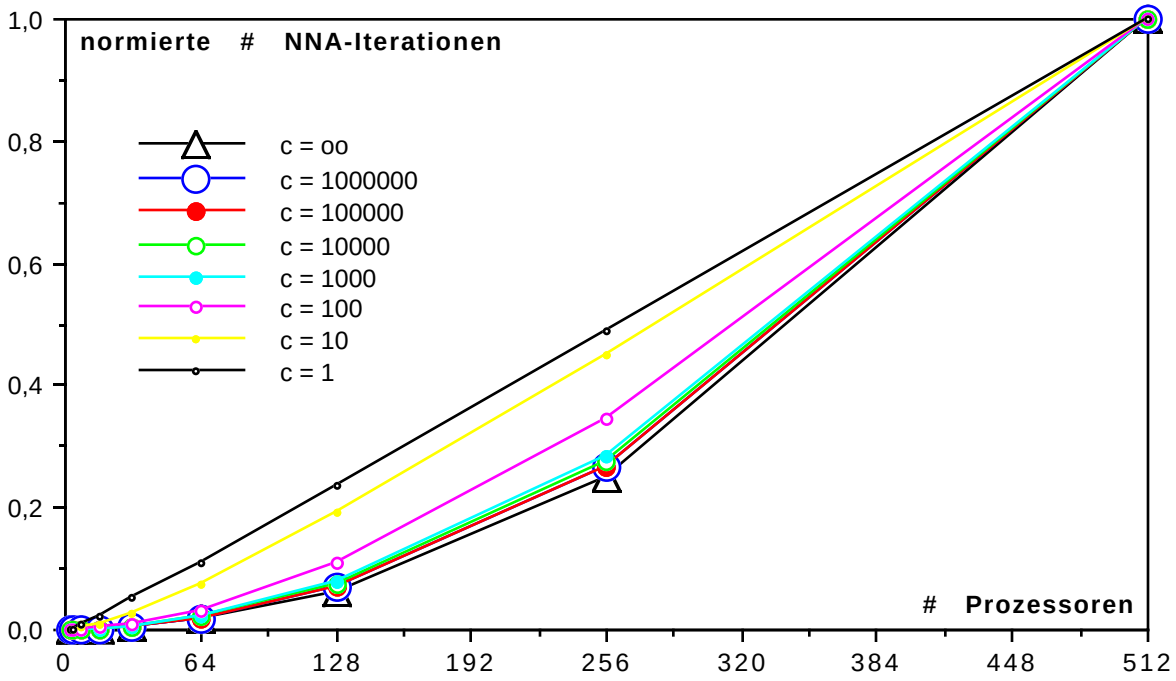

Abb. 4.5: Abhängigkeit der Nachbarschafts-Mittelung (NNA) von der initialen Lastdifferenz $L_{sum} = c \cdot P$ *dargestellt durch die auf 1 normierte Anzahl von Iterationen bei 512 Prozessoren*

Anzahl von NNA-Iterationen aufgetragen. Für NNA enthält eine Iteration meist mehrere Shifts. Die Meßwerte zu einer initialen Last wurden so normiert, daß sie bei 512 Prozessoren auf den Wert 1 zu liegen kommen. Trägt man statt den Iterationen die Shifts auf, dann zeigt NNA ein nicht ganz so ausgeprägtes Verhalten, da die mehrfachen Shifts pro Iteration das quadratische Verhalten verwischen. Insgesamt steigt der quadratische Anteil in der Laufzeit mit der initialen Belastung des ersten Prozessors bzw. mit Gesamtzahl von Lastelementen an. Der Grenzfall bilden die kontinuierliche Lasteinheiten.

In Abb. 4.6 sind die Simulationsergebnisse des Flüssigkeitsmodells im zweidimensionalen Torus für wachsende Prozessoranzahlen dargestellt. Als Anfangsbelastung wurde in (a) der ungünstigste Fall mit $c \cdot P$ Lasteinheiten auf nur dem ersten Prozessor angenommen (c = 5). In (b) wurde die anfängliche Prozessorbelastung mit einem Pseudozufallszahlen-Generator zwischen 0 und 100 Lasteinheiten gleichverteilt. In jeden Meßpunkt ist die mittlere Laufzeit und das Streuungsintervall für 10 unterschiedliche Ausgangsbelastungen angegeben. Das Flüssigkeitsmodell ergibt für alle drei Bedingungen (C3 bis C5) dieselben Laufzeiten.

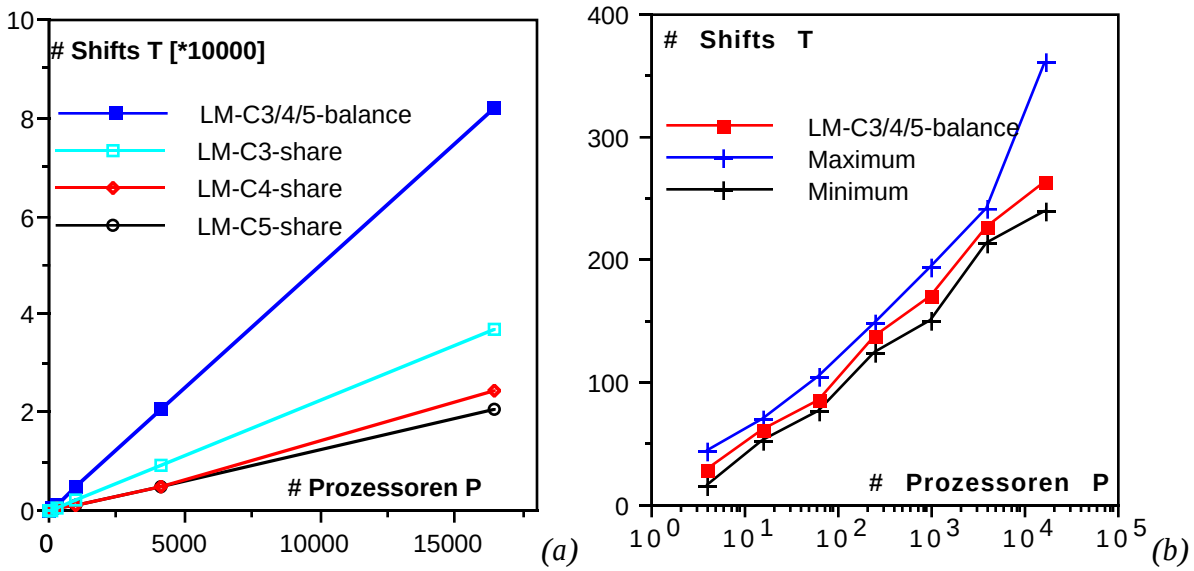

Abb. 4.6: Simulationsergebnisse des Flüssigkeitsmodells mit verschieden Shift-Bedingungen im zweidimensionalen Torus für den ungünstigsten Fall (a) und für eine zufallsverteilte Last (b)

4.4 Lokale Trigger-Mechanismen

In den letzten beiden Kapitel wurden der Lastverteilungsprozeß für sich alleine betrachtet. Zusammen mit dem B&B-Prozeß werden die beiden Prozesse ineinander verwoben, d.h. sie werden alternierend ausgeführt. Nun stellt sich die Frage, welcher Prozeß wann weiterarbeiten soll. Für SIMD-Rechner wird die Fragestellung noch wichtiger, da die Prozessoren gezwungen sind alle gleichzeitig denselben Prozeß zu bearbeiten. In dem parallelen B&B bestimmt nach jeder Iteration ein Trigger-Mechanismus, ob eine Lastverteilung notwendig und sinnvoll ist oder nicht.

In den bisherigen Vorarbeiten wurden zwei Typen von Trigger für synchrone Lastverteilung verwendet [Karypis92, Mahanti93, Powley93]. Erstens, der statische Trigger S^x zeigt notwendige Lastverteilung an, falls die Anzahl der aktiven Prozessoren A unter einen festen Bruchteil x aller verfügbaren Prozessoren fällt, d.h. falls $A \leq x \cdot P$ gilt. Zweitens, der dynamische Trigger: D^P oder D^K sind anpassungsfähig und initiieren die Lastverteilung, falls der Gewinn durch die besser verteilte Last größer ist als der Verlust durch die dafür notwendige Lastverteilung. Alle diese Trigger haben den Nachteil, daß sie globale Information verwenden, denn in beiden Fällen wird zwischen jeder B&B-Iteration die Anzahl von aktiven Prozessoren berechnet.

Hier wird ein periodischer Trigger P^f verwendet. Die Last wird mit der Frequenz f verteilt, d.h. jede $1/f$-te B&B-Iteration findet ein Lastverteilungsschritt statt. Dieser Trigger verwendet keine globale Information und kann sich damit

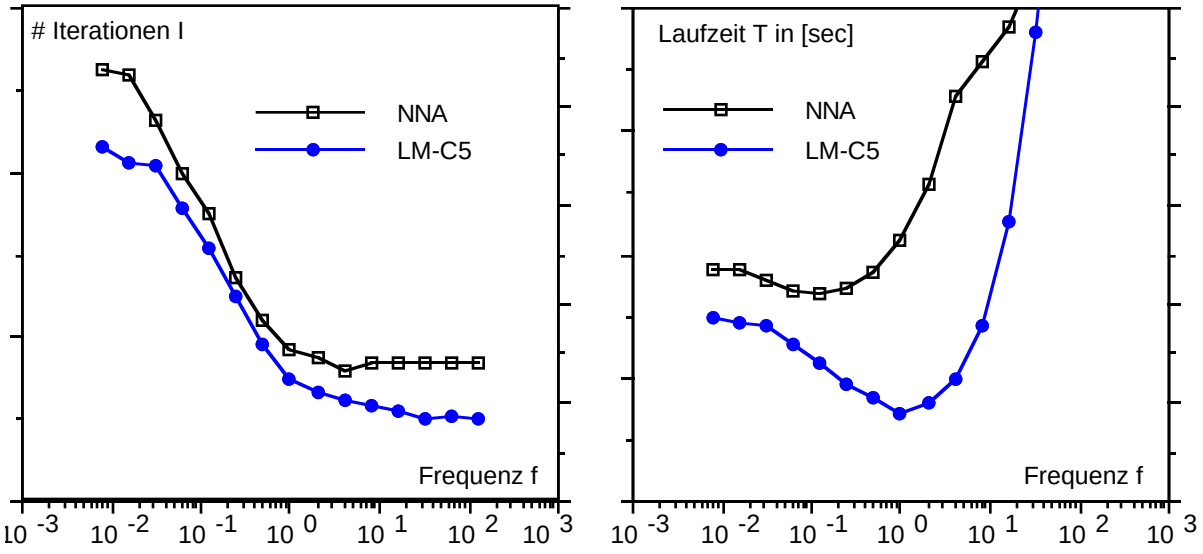

Abb. 4.7: Abhängigkeit der Anzahl Iterationen I *und der Gesamtlaufzeit* T *von der Trigger-Frequenz* f *beim Branch-and-bound mit dem Flüssigkeitsmodell (LM-C5) bzw. mit der Nachbarschafts-Mittelung (NNA) als Lastverteilung*

nicht dem Systemzustand anpassen. Da in SIMD-Rechner nur ein Befehlsstrom existiert, wird auf allen Prozessoren die Lastverteilung gleichzeitig durch den Trigger-Mechanismus initiiert. Die Auswahl des Triggers ist daher abhängig von der Leistung des globalen Informationsmechanismus (Reduktion), welche wiederum von dem Durchmesser des Netzwerkes abhängt.

Bei Verwendung des periodischen Triggers muß die Frequenz der Lastverteilung bestimmt werden. Für die hier betrachtete Problemdomäne (Kapitel 1.5) wurde eine Reihe von Probleminstanzen ausgewählt und mit unterschiedlichen Trigger-Frequenzen gelöst. Die mittlere Anzahl von B&B-Iterationen und die mittlere Laufzeit sind in Abb. 4.7 in unskaliertem Maßstab dargestellt. Wie erwartet nimmt die Anzahl von Iterationen mit zunehmender Frequenz ab und nähert sich einem Optimum. Die Gesamtlaufzeit besitzt ein Minimum. Für NNA bzw. LM sind die optimalen Frequenzen $f_{NNA} = 0{,}125$ und $f_{LM} = 1$, in Shifts pro B&B-Iteration. Sicherlich hängen die optimalen Frequenzen von dem Verhältnis von Kommunikations- und Berechnungszeit ab und müssen daher für andere Anwendungsdomänen und zugrundeliegende Rechner neu bestimmt werden.

4.5 Experimentelle Ergebnisse

Um die Effizienz des LM für paralleles B&B aufzuzeigen, wird es auf eine reale Problemdomäne angewendet. Dazu werden NP-harte Probleme der statischen, non-operationalen Belegungsplanung aus Kapitel 1.5 gelöst.

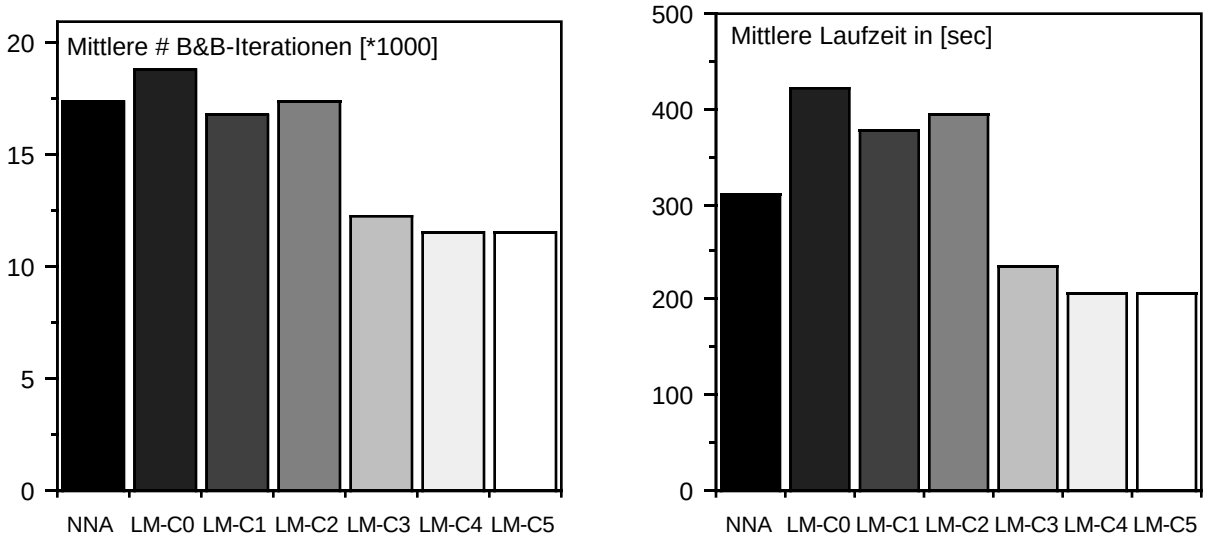

Abb. 4.8: Über 20 Benchmark-Probleme gemittelte Anzahl von Iterationen I *und Laufzeit* T *des B&B mit verschiedenen Lastverteilungsmethoden*

Vor Eintritt in die Hauptschleife des synchronen B&B wird die selektive Initialisierung aus Kapitel 3.2.3 ausgeführt, um die Prozessoren effizient mit Knoten zu versorgen. Die Auswahl der Knoten selbst verfolgt eine Tiefensuche. Nach jeder B&B-Iteration wird eine der oben beschriebenen Lastverteilungsmethoden aufgerufen. Ein Ablaufschema für den Gesamtalgorithmus ist in Abb. 1.6 gegeben. Um einen fairen Vergleich zu ermöglichen, muß die Anzahl der zu expandierenden Knoten für die verschiedenen Lastverteilungsmethoden konstant gehalten werden. Dazu wird nach allen Lösungen gesucht und ein vorgegebener Beschneidungswert verwendet. (Hier ergibt sich dieser Wert aus der zu Beginn berechneten heuristischen Lösung.)

Für die Experimente wurde der SIMD-Rechner MasPar MP-1 mit 16.384 Prozessoren in einem 2-dimensionalen Torus verwendet. Insgesamt wurden 20 Probleminstanzen mit jeweils 10^6 bis 10^8 zu expandierenden Knoten gelöst (die mittleren und schweren Benchmark-Probleme aus Kapitel 7.1). In Abb. 4.8 ist die mittlere Anzahl von B&B-Iterationen und die mittlere Laufzeit für verschiedene Lastverteilungsmethoden aufgetragen. Für den periodischen Trigger wurden die optimalen Frequenzen f_{NNA} und f_{LM} aus Kapitel 4.4 verwendet.

Die verschiedenen Shift-Bedingungen aus der Tabelle 4.1 zeigen sehr unterschiedliche Verhaltensweisen in den experimentellen Ergebnissen aus Abb. 4.8. Alle Methoden des Flüssigkeitsmodells, mit Bedingungen die nur das Ziel Load-sharing verfolgen (C0-C2), sind relativ langsam. Im Gegensatz dazu ist das LM mit Bedingungen die das Ziel Load-balancing verfolgen (C3-C5) sehr effizient. Jede dieser Lastverteilungsvarianten ist besser als NNA.

4.6 Zusammenfassung

Die Realisierung eines einfachen Flüssigkeitsmodells führt zu einer Reihe effizienter und skalierbarer dynamischen Lastverteilungsmethoden (LM-C0 bis C5). Dies wird vor allem durch die strenge Lokalität des Ansatzes sowie durch die Ausnutzung der gegebenen engen Kopplung der Prozessoren erreicht. Dabei macht die enge Kopplung eine Betrachtung der Kommunikationsmenge als realistischeres Aufwandsmaß gegenüber den Kommunikationsschritten notwendig.

Das Flüssigkeitsmodell wurde mit der bekannten, aus dem Asynchronen übertragenen, Nachbarschafts-Mittelung (NNA) verglichen. Analytisch wurde gezeigt, daß der in der Prozessoranzahl für mehrdimensionale Tori bisher quadratische Aufwand durch das Flüssigkeitsmodell auf einen linearen Aufwand reduziert wird. Auch bezüglich den Konstanten ist der hier untersuchte Ansatz (LM-C5) signifikant effizienter. Es konnte eine Effizienzverbesserung um 30 % für den parallelen Branch-and-bound-Algorithmus erreicht werden.

Neben der Effizienz des Ansatzes liegt ein weiterer Vorteil in der Kombination von Load-sharing mit Load-balancing. Die einfachste Shift-Bedingung, welche beide dieser Ziele anstrebt (LM-C5), führt zu den besten Gesamtlaufzeiten. Das Flüssigkeitsmodell verfolgt die Aufgabe der Wiederbeschäftigung mit höchster Priorität bevor die Aufgabe des Lastausgleichs in Angriff genommen wird. NNA erreicht das Ziel Load-sharing nur als Konsequenz von Load-balancing. Insbesondere für das parallele B&B zeigt sich eine Priorisierung von Load-sharing als sehr effizient.

Das Flüssigkeitsmodell ist unabhängig von dem Trigger-Mechanismus, der die Lastverteilung initialisiert. Der eingeführte periodische Trigger verwendet keine globale Information und ist daher nicht adaptiv. Die Wahl des besten Trigger-Mechanismus hängt von dem Verhältnis zwischen Kommunikations- und Berechnungszeit ab.

Eine weiterführende Arbeit auf diesem Gebiet stellt die Untersuchung der asynchronen Verarbeitung des Flüssigkeitsmodells dar. Für die üblichen MIMD-Rechner ist das Verhältnis von Aufbau- zu Übertragungszeit für Kommunikation größer als bei SIMD-Maschinen. Der periodische Trigger muß angepaßt werden, und dies wird sicherlich das Gesamtlaufzeitverhalten beeinflussen. Zusätzlich sollte das Flüssigkeitsmodell in weiteren Problemdomänen des parallelen B&B angewendet werden. Genauso wie dies für Ringe und mehrdimensionale Tori gezeigt wurde, kann der Ansatz auch für weitere Topologien, wie z.B. Hyperkuben, eingesetzt werden.

5. Implizite Lastverteilung

In diesem Kapitel wird nach einer kurzen Einleitung in Kapitel 5.1 das Konzept der k-Expansion vorgestellt. Es handelt sich um einen neuen Grundmechanismus zur Ausführung des parallelen Branch-and-bound-Algorithmus (B&B) der in Kapitel 5.2 angegeben und bewertet wird. Die k-Expansion selbst kann auf zwei unterschiedliche Weisen realisiert werden, welche in Kapitel 5.3 beschrieben sind. Verschiedene Varianten der Ausführungen in Kapitel 5.4 bestimmen, auf welche Art die k-Expansion in dem B&B eingesetzt wird. Schließlich werden in Kapitel 5.5 die erzielten experimentellen Ergebnisse zusammengestellt.

5.1 Einleitung

Bei der impliziten Lastverteilung werden die gleichen Ziele verfolgt wie bei der expliziten (statischen bzw. dynamischen) Lastverteilung: Das Load-balancing und das Load-sharing mit möglichst geringem Kommunikationsaufwand. Aber um diese Ziele zu erreichen, wird hier eine grundsätzlich andere Methode gewählt. Bei den expliziten Ansätzen wird für die Lastverteilung ein eigenständiger Prozeß als Gegenspieler zu dem B&B-Prozeß angenommen. Hier werden nun die beiden Prozesse der Lastverteilung und des B&B-Algorithmus in einen Prozeß integriert.

Zur Integration der beiden Prozesse wird ein neuer Grundmechanismus des B&B-Algorithmus für feinkörnige Verarbeitung auf SIMD-Architekturen angewendet. Er vermeidet die schwierige Aufgabe der Lastverteilung durch Einführung des Schemas der k-Expansion für parallele B&B. Die k-Expansion verteilt die aufkommende Last implizit durch weiteres Verzweigen des Suchbaums und macht einen zusätzlichen Mechanismus für die Lastverteilung überflüssig. Diese Expansion und die darauffolgende Auswertung der Knoten können parallel mit allen Prozessoren durchgeführt werden, so daß sich automatisch eine Auslastung der Prozessoren ergibt. Dabei wird angenommen, daß der Aufwand zur Knotengenerierung im Vergleich zur Knotenauswertung

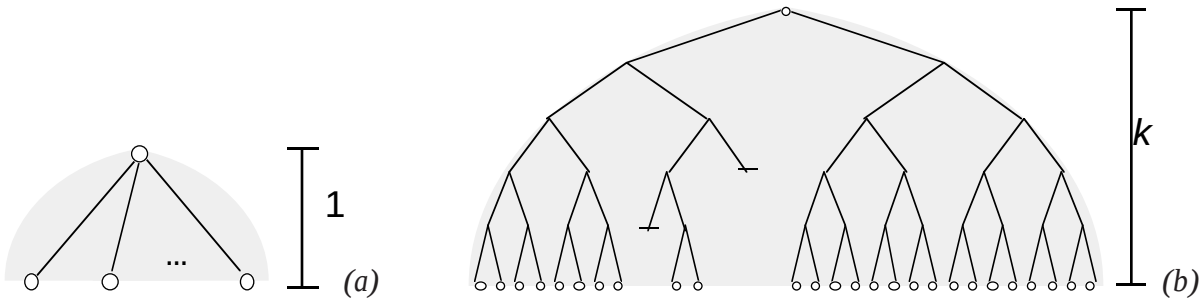

Abb. 5.1: Schema der 1-Expansion (a) und der k-Expansion (b) für Suchbäume

gering ist. Zusammenfassend ist die implizite Lastverteilung eine Verschmelzung des B&B mit dem Lastverteilungsprozeß, so daß durch die veränderte Bearbeitung des Suchbaums die Last automatisch auf die Prozessoren verteilt wird.

Als Berechnungsmodell wird eine feinkörnige SIMD-Maschine angenommen (Kapitel 1.3). Die Verbindungstopologie der Prozessoren ist bei dem hier beschriebenen Algorithmus unerheblich. Dafür werden zwei, speziell bei dieser Architektur unterstützte Mechanismen ausgenutzt. Der erste Mechanismus ist die Berechnung des Minimums über einen skalaren Wert in jedem Prozessor. Die Minimumbildung kann durch eine sogenannte *Reduktion* effizient durchgeführt werden. Zum Beispiel sind bei einer PRAM mit P Prozessoren dafür $O(\log P)$ Schritte notwendig [Akl89]. Ein weiterer notwendiger Mechanismus ist das Versenden eines Datensatzes an alle Prozessoren *(Broadcast)*. Bei einer Busverbindung der Prozessoren benötigt das Versenden nur einen konstanten asymptotischen Zeitaufwand $O(1)$.

Zur Vereinfachung der Darstellung werden hier nur balancierte Suchbäume betrachtet. Die Blätter des Suchbaums (Lösungsknoten) liegen alle in derselben Tiefe d. Durch Modifikation des Grundschemas können auch nichtbalancierte Suchbäume bearbeitet werden.

5.2 Grundkonzept

In diesem Unterkapitel wird das neue Konzept der k-Expansion in Suchbäumen dargestellt und in den üblichen B&B-Algorithmus eingebunden. Danach wird die parallele Abarbeitung der k-Expansion diskutiert. Schließlich folgt eine Bewertung des vorgestellten Konzepts.

5.2.1 Branch-and-bound mit k-Expansion

Die übliche Iteration des sequentiellen B&B arbeitet nach dem folgenden Schema: Mit Hilfe einer bestimmten Strategie wird ein Knoten aus der Datenstruktur OPEN ausgewählt und expandiert. Es wird nun angenommen, daß

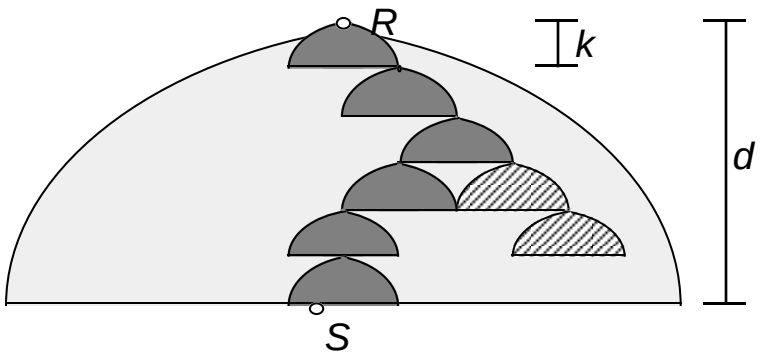

Abb. 5.2: Schema der Abarbeitung eines Suchbaums der Tiefe d *durch das Branch-and-bound mit k-Expansion vom Wurzelknoten* R *bis zum optimalen Lösungsknoten* S *inkl. einer Sackgasse*

dieser Knoten in der Tiefe *i* des zu bearbeitenden Suchbaums angesiedelt sei. Um diesen Knoten zu expandieren, werden alle seine Nachfolger in der Tiefe $i + 1$ generiert (Abb. 5.1(a)). Die Anzahl von Nachfolgern hängt von dem Verzweigungsfaktor des Suchbaums ab. Nach Auswertung der Nachfolger werden diese schließlich in OPEN eingefügt oder nicht mehr weiter betrachtet (Kapitel 1.2).

Sei nun ein Suchbaum mit einer maximalen Tiefe *d* und ein beliebiger Knoten in der Tiefe *i* des Suchbaums gegeben. Dann generiert die *k-Expansion* alle Nachfolger dieses Knotens in der Tiefe $i + k$ (Abb. 5.1(b)). Falls kein Knoten mit dieser Tiefe existiert, da unter Umständen $i + k$ die maximale Tiefe *d* des Baums überschreitet, dann werden statt dessen die entsprechenden Knoten in Tiefe *d* erzeugt. Damit stellt die sonst übliche Iteration des B&B einen Spezialfall der k-Expansion dar, wobei $k = 1$ gewählt bzw. eine 1-Expansion durchgeführt wird.

Bei dem *Branch-and-bound mit k-Expansion* wird in der üblichen Iteration die 1-Expansion durch eine k-Expansion ersetzt. Nachdem ein Knoten aus OPEN ausgewählt wurde, wird eine k-Expansion durchgeführt. Dabei werden die Nachfolger in der Tiefe $i + k$ statt in der Tiefe $i + 1$ erzeugt. Das heißt, eine Iteration mit k-Expansion arbeitet einen Teilbaum der Tiefe *k* ab. Die Generierung der Nachfolger bzw. die Blätter des Teilbaums an sich wird in Kapitel 5.3 beschrieben. Alle anderen Mechanismen des B&B bleiben unverändert.

In Abb. 5.2 ist die Vorgehensweise im Gesamten für einen Suchbaum schematisch dargestellt. Beginnend von dem globalen Wurzelknoten *R* werden in jeder Iteration die Blätter von Teilbäumen mit Tiefe *k* erzeugt und ausgewertet. Natürlich kann auch hier der Algorithmus in Sackgassen (lokale Minima der Bewertungsfunktion) hineingeraten und muß auf einen Vorgängerknoten zurücksetzen. Nach einigen Iterationen generiert der Algorithmus auch eine Reihe von Blätter des ursprünglichen Suchbaums, d.h. Lösungsknoten des Problems. Der letzte Reduktionsschritt berechnet einen

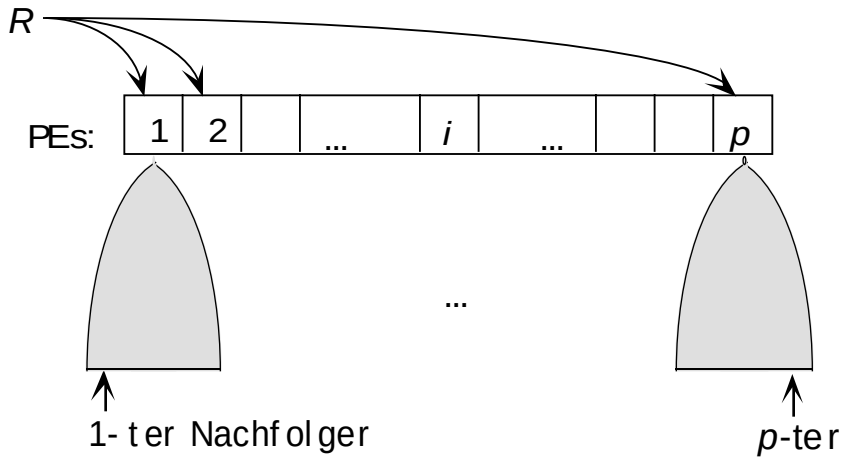

Abb. 5.3: Schema der parallelen k-Expansion mit P *Prozessoren (PEs), ausgehend von dem Wurzelknoten* R *des zu expandierenden Teilbaums*

minimalen Lösungsknoten *S*. Falls die Bewertung von *S* besser als alle anderen Werte der noch verbleibenden Knoten ist, dann stellt *S* eine optimale Lösung des Problems dar und kann ausgegeben werden.

5.2.2 k-Expansion mit *P* Prozessoren

Für die Ausführung des B&B mit mehreren Prozessoren kann die k-Expansion einfach parallelisiert werden. In Abb. 5.3 ist die Vorgehensweise schematisch dargestellt. Der jeweilige Wurzelknoten *R* des zu expandierenden Teilbaums wird an alle Prozessoren versandt. Auf eine noch festzulegende Art wird der Teilbaum expandiert, so daß der *i*-te Prozessor den *i*-ten Knoten in der Tiefe *k* des Teilbaums erhält (Kapitel 5.3). Außerdem wird die Teilbaumtiefe *k* so gewählt, daß die Prozessoren maximal ausgelastet werden (dazu Kapitel 5.3.2).

Wird dieser Mechanismus in den B&B integriert, so beginnt die erste Iteration des B&B mit dem (globalen) Wurzelknoten des Suchbaums. Dann folgen weitere Iterationen. Zu Anfang einer Iteration wird der Knoten mit der besten Bewertung, die lokale Wurzel (Abb. 5.4(a)), an alle Prozessoren über ein Broadcast versandt (b). Er repräsentiert den lokalen Wurzelknoten eines Teilbaums. Jeder Prozessor generiert seinen Nachfolger in der Tiefe *k* unter Verwendung dieses lokalen Wurzelknotens (c). Alle Nachfolgerknoten sind in der Tiefe *k* ab der lokalen Wurzel angesiedelt und sind paarweise verschieden. Für bestimmte Methoden zur k-Expansion kann diese Generierung in einem einzigen Schritt erfolgen. Jeder generierte Nachfolger kann mit Hilfe der Bewertungsfunktion parallel ausgewertet und, falls notwendig, in die Datenstruktur OPEN parallel eingefügt werden (d). In einem Reduktionsschritt wird aus allen lokalen OPEN-Mengen derjenige Knoten herausgesucht, der die beste (kleinste) Bewertung erhalten hat (e). In den nächsten Iterationen des B&B wird der Expansionsschritt mit dem jeweils besten Knoten wiederholt. Insgesamt bearbeitet der Algorithmus den Suchbaum in der gleichen Art wie der

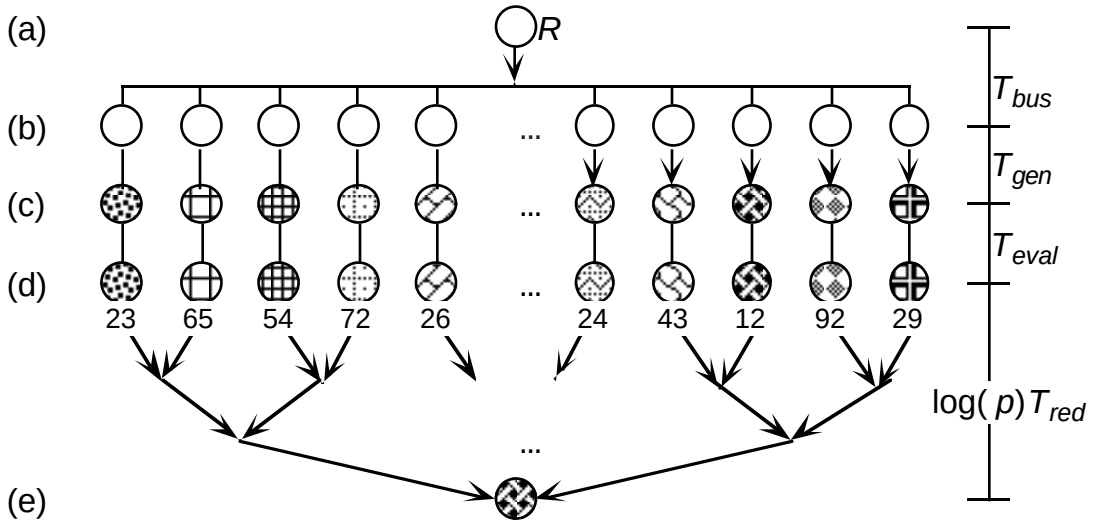

Abb. 5.4: Ablauf und Aufwand einer Iteration des Branch-and-bound mit k-Expansion in vier Schritten: Versenden des besten Knotens (a)→(b), Expansion (b)→(c), Bewertung (c)→(d) und Reduktion (d)→(e)

herkömmliche B&B mit dem Unterschied, daß der jeweils beste Knoten um *k* Schritte expandiert wird statt um einen.

Man kann den B&B mit k-Expansion auch als eine Anpassung des Suchbaums an die Prozessoranzahl betrachten. Wenn sichergestellt wäre, daß der Verzweigungsfaktor des Suchbaums ähnlich zu der Prozessoranzahl ist, dann würde eine Parallelisierung des B&B auf dem Expansionslevel möglich werden. In anderen Worten: Die Erzeugung und Auswertung der Nachfolger könnte dann vollständig parallel ausgeführt werden. Durch die Einführung der k-Expansion wird dieser Effekt erzielt, da der Verzweigungsfaktor *b* auf etwa die Anzahl der Prozessoren *P* vergrößert wird.

5.2.3 Auswertung

Das Konzept der k-Expansion selbst sagt zunächst nichts über die Parallelisierung des Algorithmus aus. Die k-Expansion ist auch in einem rein sequentiellen B&B denkbar. Aber die Vorteile der k-Expansion kommen erst bei der parallelen Verarbeitung zum Tragen. Im folgenden wird also immer von einer k-Expansion auf Parallelrechner ausgegangen.

In jeder Iteration des B&B werden die Prozessoren durch die k-Expansion automatisch ausgelastet, und es ist keine explizite Lastverteilung notwendig. Außerdem wird der globale beste Knoten durch die Reduktion bestimmt. Im Gegensatz zu den meisten anderen parallelen B&B führt dieser Algorithmus eine globale Suchstrategie auf der Ebene der expandierten Teilbäume durch. Andererseits ist der Mehraufwand resultierend aus der k-Expansion klein. Er besteht aus mehreren Punkten:

- Die globale Auswahl des nächsten lokalen Wurzelknotens (Reduktion): Die Knotenauswahl entspricht einer Minimumsbildung in der globalen OPEN-Menge. Sie ist durch den Reduktionsschritt gegenüber dem sequentiellen Algorithmus sicherlich ein zusätzlicher Aufwand. Damit bleibt der zusätzliche Aufwand zwar abhängig von der Prozessoranzahl, aber insbesondere auf SIMD-Rechnern wird diese Operation durch sehr effiziente Standardfunktionen abgefangen.
- Die Verteilung der lokalen Wurzel an alle Prozessoren (Broadcast): Falls eine Busverbindung zwischen den Prozessoren verfügbar ist, kann die Wurzel in konstanter Zeit versandt werden. Bei allen SIMD-Architekturen ist ein solcher Bus vorhanden, allein schon um den einheitlichen Befehlsstrom an alle Prozessoren weiterzuleiten.
- Die k-Expansion an sich: Sie kann unter Umständen etwas aufwendiger sein als die Erzeugung der direkten Nachfolger eines Knotens. Asymptotisch betrachtet geht die Prozessoranzahl P logarithmisch in den Aufwand der k-Expansion ein. Aber die direkte Generierung der Nachfolger in Tiefe k benötigt keine Kommunikation zwischen den Prozessoren. Sie ist daher neben der Initialisierung auch für den wiederholten Einsatz auf Parallelrechnern gut geeignet. Für die asymptotische Betrachtung kann durchaus der k-Expansionsschritt als elementar betrachtet werden.[1]
- Die Wartezeiten auf Grund von ungünstigen Verzweigungsfaktoren: Dieser Effekt hängt von dem gewählten Schema der k-Expansion ab und wird in einem späteren Kapitel diskutiert.

5.3 Realisierung

In der obigen Beschreibung des B&B mit k-Expansion wurde die Realisierung des Konzepts der k-Expansion selbst übersprungen. Bei der Realisierung der k-Expansion kann man mehrere Ausprägungen unterscheiden. Welche davon angewendet werden kann, hängt von der Struktur des Suchbaums ab. Zwei mögliche Ausprägungen werden in den folgenden Unterkapiteln beschrieben und ausgewertet. Für eine der beiden wird außerdem die Wahl des freien Parameters k diskutiert.

5.3.1 Direkte k-Expansion

In diesem Unterkapitel wird eine Möglichkeit zur Realisierung der k-Expansion dargestellt und untersucht. Diese Ausprägung entspricht der direkten Initialisierung aus Kapitel 3.2.4 und soll daher hier nur kurz beschrieben werden.

---

[1] Diese Annahme ist analog zu der als konstant angenommenen Wortgröße der Prozessoren in Parallelrechnern, welche allein schon durch darzustellenden Prozessorindizes logarithmisch in P wächst.

Diese Expansionsmethode ist nur anwendbar, falls der Struktur des Suchbaums im voraus bekannt ist. In diesem Fall kann eine Funktion zur Knotengenerierung aufgestellt werden. Sie generiert den *j*-ten Knoten in einer beliebigen Tiefe *k* des Suchbaums, ohne seine Vorgänger zu betrachten. Dieses Vorgehen wird hier "direkte" Knotengenerierung genannt.

Die Anwendung der direkten Knotengenerierung führt zu der *direkten* k-Expansion (DE). Bei *P* Prozessoren berechnet der Prozessor *i* ausgehend von der Wurzel *R* des Teilbaums den *j*-ten Nachfolger in der Tiefe *k* mit $i = j \bmod P$. Da die Struktur des Suchbaums als bekannt vorausgesetzt wird, kann die Generation eines Knotens in einem Schritt geschehen (Abb. 3.6). Um die Korrektheit der k-Expansion zu gewährleisten, liegen alle generierten Knoten in derselben Tiefe des Suchbaums. Außerdem müssen alle Blätter des Teilbaums generiert werden, so daß u.U. mehrere Knoten auf einen Prozessor entfallen. Zusätzlich ist eine hohe Auslastung für den gesamten Suchbaum erwünscht. Daher wird die Tiefe so gewählt, daß die Wartezeiten der Prozessoren minimiert werden (Kapitel 5.3.2).

Für eine Aufwandsanalyse des B&B mit direkter k-Expansion werden einige vereinfachende Annahmen gemacht. Die problemabhängige Entwicklung des Algorithmus, wie z.B. das Beschneiden von Zweigen, während der Abarbeitung des Suchbaums soll eingeschränkt werden. Dazu wird angenommen, daß der gesamte Suchbaum vom B&B abgearbeitet werden muß, um das Optimum zu finden. Wir gehen also von einer extrem ungünstigen und aufwendigen Problemstellung aus. Zusätzlich wird angenommen, daß der Suchbaum balanciert ist. Das heißt, alle zulässigen Lösungsknoten sind in derselben Tiefe des Suchbaums angesiedelt. Die letzte Einschränkung an die Gestalt des Suchraums ist zum Beispiel bei vielen Belegungsplanungsproblemen erfüllt (Kapitel 7.2.1). Um die benötigte Zeit für eine B&B-Iteration mit k-Expansion zu modellieren werden die folgenden Zeitkonstanten eingeführt:

T_{red} (mittlere) Zeit für einen Reduktionsschritt. Insgesamt sind mindestens $\Omega(\log P)$ Reduktionsschritte notwendig, um das Minimum von Werten zu berechnen, welche auf *P* Prozessoren verteilt sind.

T_{send} (mittlere) Zeit zum Versenden eines Knoten von dem Vorrechner an alle Prozessoren. Falls ein Bus verfügbar ist, dann kann diese Zeit als konstant angenommen werden. Andernfalls sind $\Omega(\log P)$ Kommunikationsschritte notwendig.

T_{gen} (mittlere) Zeit zur Generierung eines Nachfolgers für einen beliebigen Knoten durch die Expansionsfunktion.

T_{eval} (mittlere) Zeit zur Berechnung der heuristischen Kostenabschätzung (untere Schranke) für einen Knoten durch die Bewertungsfunktion.

Zur Abschätzung des Gesamtaufwands des Algorithmus betrachten wir einen einzelnen Teilbaum. Da hier zur Analyse angenommen wird, daß der Suchbaum vollständig abgearbeitet werden muß, wird jeder Teilbaum des parallelen Algorithmus auch von dem sequentiellen Algorithmus bearbeitet. Es kann also

von dem Aufwand zur Bearbeitung eines Teilbaums auf den Gesamtaufwand geschlossen werden.

Der Aufwand zur parallelen Bearbeitung eines Teilbaums wird nun mit der sequentiellen Bearbeitung verglichen. Dabei ist die Größe der Teilbäume bestimmt durch eine Iteration des B&B mit k-Expansion. Ein Baum mit konstantem Verzweigungsfaktor $b > 0$ hat in der Tiefe i genau b^i Knoten. Die Anzahl $N(b, k)$ aller Knoten innerhalb eines solchen Teilbaums berechnet sich bei fester Tiefe k des Teilbaums aus der geschlossenen Form der geometrischen Reihe. Außerdem kann man annehmen, daß k möglichst nahe bei $\log_b(P)$ gewählt wird. Damit ergibt sich:

$$N(b, k) := \sum_{i=0}^{k} b^i = \frac{b^{k+1} - 1}{b - 1} \approx \frac{P\,b - 1}{b - 1} \tag{5.1}$$

Um mit nur einem Prozessor einen solchen Teilbaum vollständig abzuarbeiten, ist für jeden Knoten (bis auf die Wurzel) eine Bearbeitung notwendig. Dabei beinhaltet eine Bearbeitung die Generierung und Auswertung des Knotens mit Hilfe der Bewertungsfunktion. Für diese beiden Aktionen wurden die konstanten Zeiten von T_{gen} bzw. T_{eval} angenommen. Der Zeitaufwand $T(P)$ bei $P = 1$ Prozessoren ist, abhängig von der Anzahl von Iterationen, also

$$T(1) = (N(b, k) - 1)\cdot(T_{gen} + T_{eval}) \tag{5.2}$$

Im Gegensatz dazu wird bei der parallelen k-Expansion mit $P > 1$ Prozessoren zuerst der beste Knoten aus der letzten Iteration (Teilbaum) an alle Prozessoren in Zeit T_{send} versandt. Dann benötigen die Prozessoren abhängig von dem Verzweigungsfaktor b und der gewählten Tiefe des Teilbaums für die parallele Bearbeitung der Blätter des Teilbaums einen oder mehrere Generierungs- und Auswertungsschritte. Zusätzlich werden noch $\log(P)$ Minimumbildungen mit Aufwand T_{red} vorgenommen. Insgesamt ergibt sich ein Zeitaufwand T_{DE} der parallelen Auswertung eines Teilbaums mit direkter k-Expansion von:

$$T_{\mathrm{DE}}(P) = \log(P)T_{\mathrm{red}} + T_{\mathrm{send}} + \left\lceil \frac{b^k}{P} \right\rceil (T_{\mathrm{gen}} + T_{\mathrm{eval}}) \tag{5.3}$$

Um die Beschleunigung dieser parallelen B&B-Methode zu analysieren, wird der Aufwand zur Bearbeitung eines Teilbaums mit Verzweigungsfaktor b herangezogen. Als Beschleunigung $S(P)$ bei P Prozessoren wird das Laufzeitverhältnis des sequentiellen und parallelen Algorithmus angenommen: $S(P) = T(1)/T(P)$. Die asymptotisch erreichbare Beschleunigung S_{DE} durch den parallelen B&B mit direkter k-Expansion ist damit gegeben durch

$$S_{\mathrm{DE}}(P) = \frac{T(1)}{T_{\mathrm{DE}}(P)} = \mathrm{O}\left(\frac{P}{\log(P)}\right) \tag{5.4}$$

Der parallele B&B mit k-Expansion erbringt nach der obigen Analyse natürlich nicht den idealen Wert der maximalen Beschleunigung S'(P) = P. Diese Beschleunigung ist auf realen Rechnern generell nicht zu erreichen, da ein Mehraufwand durch die notwendige Kommunikation und Koordination hinzu kommt. In unserem Fall wirkt sich vor allem der Reduktionsschritt verlangsamend aus. Aber die asymptotische Beschleunigung liegt deutlich über dem Wert aus Minsky's Vermutung. Diese pessimistische Vermutung sagt aus, daß im allgemeinen durch Parallelität eine Beschleunigung nur um den Faktor log$P$ zu erreichen ist [Gonauser89].

Neben der oben durchgeführten Beschleunigungsanalyse gibt es noch weitere, eher qualitative Kriterien. Die Ausprägung der direkten k-Expansion für den parallelen B&B läßt sich wie folgt bewerten:

– Diese Ausprägung ist nur bei strukturierten Suchbäumen anwendbar, da ansonsten die Nachfolger in der Tiefe k nicht direkt generiert werden können. Aber die Struktur der Suchbäume ist für viele Problemstellungen, wie z.B. bei der Belegungsplanung, bekannt.

– Während der Bearbeitung eines Teilbaums werden ungünstige Zwischenknoten nicht beschnitten, denn der Expansionsschritt generiert die Blätter des Teilbaums direkt. Knoten zwischen der lokalen Wurzel und den Blättern werden nicht bewertet. Einige Prozessoren erzeugen und bewerten damit ungünstige Blätter, die erst am Ende der Iteration verworfen werden. Falls in Tiefe i der Teilbäume b^{i-1} Knoten beschnitten werden, dann bearbeiten die direkte k-Expansion und die 1-Expansion gleich viele Knoten.

\+ Die k-Expansion kann unter Umständen etwas aufwendiger sein als die Erzeugung der direkten Nachfolger eines Knotens. Im Vergleich zu der zweiten Ausprägung, der selektiven k-Expansion (Kapitel 5.3.3), ist die direkte k-Expansion sehr schnell auszuführen.

\+ Durch die direkte Generierung von Knoten in der Tiefe k werden einige Suchbaumebenen übersprungen. Die Knoten in diesen Ebenen müssen nicht bearbeitet werden, d.h. insgesamt werden weniger Knoten des Suchbaums betrachtet als im sequentiellen Fall. Im Extremfall bei Teilbaumtiefe $k = d$ wird in Binärbäumen nur die Hälfte aller Knoten betrachtet. Daher ist bei vollständiger Suche eine "superlineare" Beschleunigung möglich.

5.3.2 Die Wahl von *k*

Für die parallele Verarbeitung des B&B ist die Anzahl von Prozessoren fest vorgegeben. Außerdem ist das Konzept der k-Expansion über die Tiefe k der expandierten Teilbäume parametrisiert. Daher kann dieser Parameter bei der direkten k-Expansion so angepaßt werden, daß eine beste Ausnutzung der vorhandenen Prozessoren möglich ist. Als ein Beispiel wird ein Teilbaum der Tiefe k mit einem konstanten Verzweigungsfaktor b betrachtet. Die Anzahl von Knoten in der Tiefe k ist b^k. Die maximale Prozessorauslastung wird erreicht,

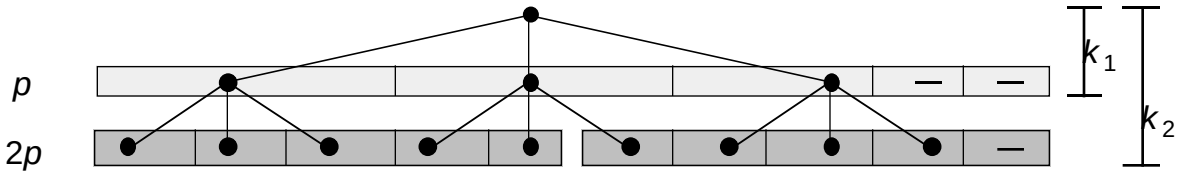

Abb. 5.5: Auswirkung unterschiedlicher Teilbaumtiefen der k-Expansion (k_1 bzw. k_2) auf die mittlere Anzahl wartender Prozessoren (2 bzw. 0,5)

wenn jeder Prozessor einen Nachfolger (Blatt des Teilbaums) generieren kann, also wenn k mit $P = b^k$ gewählt wird. Beim Invertieren dieser Gleichung wird der Parameter k ausgewertet zu $k = \log_b(P)$. Durch die Quantisierung der Baumtiefen auf ganzzahlige Werte bleiben also zwei Möglichkeiten für die konkrete Wahl von k:

$$k_1 = \lfloor \log_b(P) \rfloor \quad \text{und} \quad k_2 = \lceil \log_b(P) \rceil \tag{5.5}$$

Größere k's und damit größere Teilbäume brauchen nicht betrachtet zu werden, da damit weniger Beschneidungen vorgenommen werden können. Die Beschneidung nach den k-Expansionen sind wichtig, da sie die Anzahl der insgesamt zu betrachtenden Knoten drastisch reduzieren können.

Falls der Verzweigungsfaktor b entweder nicht konstant oder nicht im voraus bekannt ist, dann kann man sich k_1 als die maximale Tiefe vorstellen, welche weniger Knoten als verfügbare Prozessoren enthält. Dementsprechend stellt dann k_2 die minimale Tiefe dar, in welcher sich mehr als P Knoten befinden. Daraus folgt, daß bei k_1 in einer k-Expansion evtl. einige Prozessoren unbeschäftigt sind und daß bei k_2 u.U. mehrere Knoten pro Prozessor in einer k-Expansion generiert werden müssen.

In Abb. 5.5 ist ein Beispiel für die Auswirkung verschiedener Teilbaumtiefen auf die mittlere Auslastung der Prozessoren gegeben. Der gegebene Suchbaum hat den konstanten Verzweigungsfaktor drei und wird von fünf Prozessoren bearbeitet. Die ideale Anzahl von Prozessoren für diese Problemstellung wäre eine Potenz des Verzweigungsfaktors, hier also: 3, 9, 27, 81, usw. Damit würden alle Blätter der Teilbäume gleichzeitig generiert. Nun stehen aber fünf Prozessoren zur Verfügung, d.h. für die Wahl eines günstigen k bleiben also die Tiefen $k_1 = 1$ und $k_2 = 2$. Bei der Tiefe k_1 generieren drei Prozessoren jeweils einen Nachfolger, und zwei Prozessoren sind unbeschäftigt. Bei der Tiefe k_2 werden hingegen neun Nachfolger in $x = 2$ Schritten erzeugt. In dem ersten Schritt sind alle Prozessoren und in dem zweiten sind nur vier Prozessoren beschäftigt. Die mittlere Auslastung liegt für k_1 bei 60 % und für k_2 bei 90 %.

Zumeist existieren bei der Teilbaumtiefe k_1 weniger Nachfolger als Prozessoren. Die meisten Prozessoren werden genau einen und einige Prozessoren keinen Nachfolger generieren können. Die (mittlere) Anzahl I_1 unbeschäftigter Prozessoren berechnet sich aus

$$I_1 = P - b^{k_1} \tag{5.6}$$

Wählt man dagegen eine Teilbaumtiefe von k_2, dann ist ein Vielfaches an Nachfolger zu generieren als Prozessoren vorhanden sind. Die meisten Prozessoren werden also x Nachfolgerknoten erzeugen und einige wenige nur $x - 1$ Nachfolger. In einzelnen Berechnungsschritten gedacht sind in den ersten $x - 1$ Erzeugungsschritten alle Prozessoren beschäftigt, und in dem letzten Schritt stehen einige still. Dabei ergibt sich die Anzahl insgesamt notwendiger Berechnungsschritte x aus der Gesamtzahl von Nachfolger pro Prozessoranzahl mit $x = \lceil b^{k_2} / P \rceil$. Die Anzahl unbeschäftigter Prozessoren in dem letzten Schritt berechnet sich aus $x\,P - b^{k_2}$. Dieser Wert verteilt sich auf die x Iterationen, so daß im Mittel I_2 Prozessoren pro Iteration warten mit

$$I_2 = \frac{x\,P - b^{k_2}}{x} \tag{5.7}$$

Um nun die Effizienz der k-Expansion zu maximieren, müssen im Mittel möglichst wenige Prozessoren pro k-Expansionsschritt stillstehen. Abhängig von den durch die Problemstellung vorgegebenen Verzweigungsfaktor b und der Anzahl der zur Verfügung stehenden Prozessoren P wird die Tiefe k der Teilbäume gewählt nach

$$k = \begin{cases} k_1, & \text{falls } I_1 \le I_2 \\ k_2, & \text{ansonsten} \end{cases} \tag{5.8}$$

Durch eine einfache Fallunterscheidung nach $P = b^k$ bzw. $P \neq b^k$, für $k \in \mathbb{N}$, kann gezeigt werden, daß immer $I_1 \geq I_2$ gilt, d.h. die mittlere Auslastung bei k_2 ist für alle Prozessoranzahlen und Suchbaumtypen besser oder gleich als bei k_1. Demnach scheint eine endgültige Entscheidung für die größeren Teilbäume k_2 günstig.[2] Andererseits sind die kleineren Teilbäume bei k_1 von Vorteil, da die Beschneidung nach der k-Expansion öfter vorgenommen werden kann. Daher sollte bei $I_1 = I_2$ immer k_1 gewählt werden.

In Abb. 5.6 sind Ergebnisse einer Simulation gezeigt. Die mittlere Anzahl unbeschäftigter Prozessoren I ist für eine wachsende Anzahl von Prozessoren abhängig von k_1 und k_2 aufgetragen (a). Als Verzweigungsfaktor wird $b = 3$ angenommen, da man für $b = 2$ durch Fallunterscheidung $I_1 = I_2$ ableiten kann. Aus dieser Simulation wird deutlich, wie sich die mittlere Anzahl unbeschäftigter Prozessoren pro Iteration für wachsende Prozessoranzahlen verändert. Außerdem ist in (b) die mittlere Auslastung U_2 mit $U_2 = (P - I_2)/P$ für verschiedene Verzweigungsfaktoren angegeben. Die Auslastung schwankt je nach Verhältnis von b zu P zwischen 50 % und 100 %.

---

[2] Asymptotisch verschwindet für wachsendes k die mittlere Anzahl unbeschäftigter Prozessoren, da maximal die Hälfte der Prozessoren pro k-Expansion unbeschäftigt ist, aber die Anzahl Iterationen x mit k wächst.

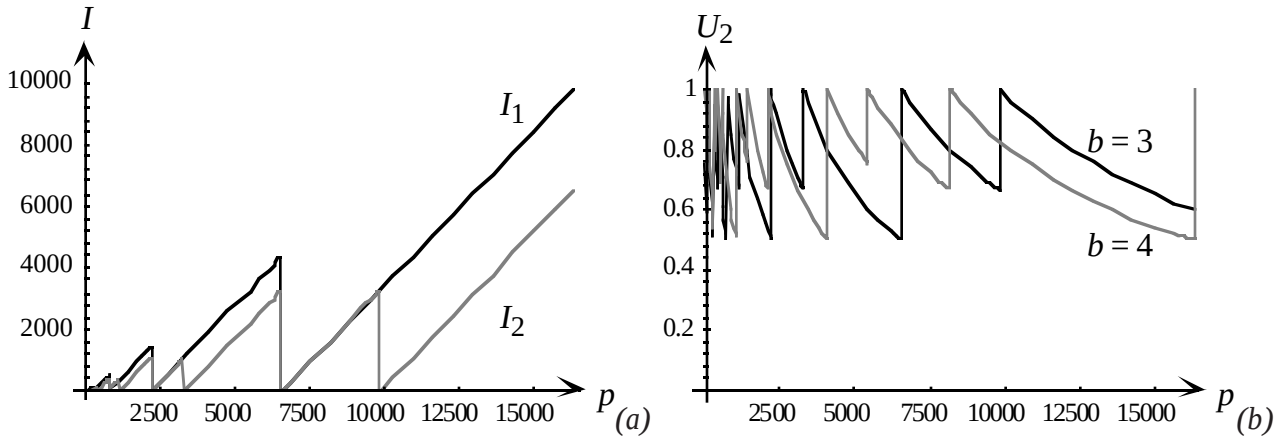

Abb. 5.6: Mittlere Anzahl I *unbeschäftigter Prozessoren bei k-Expansion mit Teilbaumtiefe* k_1 *und* k_2 *(a) und die Auslastung der Prozessoren bei der k-Expansion für Suchbäume mit Verzweigungsfaktoren* b = *3 und* b = *4 (b)*

Eine Schwierigkeit der direkten k-Expansion ergibt sich aus dem Verhältnis des Parameters *k* zu der Gesamttiefe *d* des Suchbaums. Der Parameter *k* gibt die Tiefe der Teilbäume an, die bei einem einzelnen Iterationsschritt expandiert werden. Bei entsprechender Wahl von *k* gehen die expandierten Teilbäume nicht exakt in dem Gesamtbaum auf. Der tiefste Teilbaum einer Iterationsfolge kann nur unvollständig expandiert werden, da die Tiefe des Gesamtbaums geringer ist als die Summe der Teilbaumtiefen. Andererseits muß dieser tiefste Teilbaum expandiert werden, da ansonsten die Blätter des Gesamtbaums nicht generiert würden. Dieses Problem tritt also immer dann auf, wenn *d* nicht ganzzahlig durch *k* teilbar ist ($d \bmod k \neq 0$). Zur Lösung dieses Problems werden hier drei verschiedene Ansätze kurz beschreiben.

(1) Bei der "Brute-force-Methode" wird die k-Expansion bis in die letzte Tiefe des (endlichen) Gesamtbaums durchgeführt. Dabei bleibt das obige Problem wie in Abb. 5.7 angedeutet einfach unberücksichtigt. Der dadurch entstehende Mehraufwand in der Suche ist erheblich. In jeder zusätzlichen Ebene vervielfacht sich die Gesamtzahl der Suchbaumknoten. Da die Knoten in der zusätzlichen Tiefe nicht generiert werden, bleiben statt dessen entsprechend viele Prozessoren für diese Iteration unbeschäftigt. Zum Beispiel werden in einem unbeschnittenen, vollständigen Binärbaum ($b = 2$) der Tiefe $d = 5$ bei einer Expansionstiefe von $k = 2$ insgesamt 21 Teilbäume expandiert (Abb. 5.7).

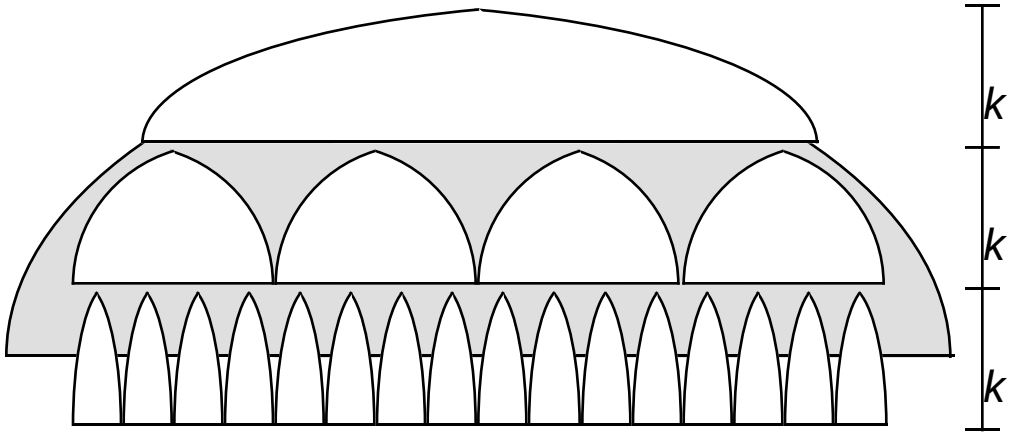

Abb. 5.7: Abarbeitung eines Suchbaums durch die k-Expansion mit fester Expansionstiefe k *bis zu tiefsten Ebene des Suchbaums*

(2) Alternativ dazu kann auch bei der k-Expansion in der letzten Ebene die Tiefe der Teilbäume angepaßt werden. Diese veränderte Tiefe *k'* wird so gewählt, daß die Blätter der Teilbäume auf Blätter des Gesamtbaums zu liegen kommen. Für *k'* muß dann $k' = k + (d \bmod k)$ gelten. Bei der Generierung der angepaßten Teilbäume werden mehrere Knoten pro Prozessor erzeugt, d.h. es sind auch mehrere Iteration zur kompletten Expansion notwendig (Abb. 5.8). Für das obige Beispiel sind mit der modifizierten letzten Tiefe nur 5 Teilbäume zu expandieren, wobei die Teilbäume der Tiefe *k'* zwei Iterationen benötigen. Insgesamt ergeben sich also 9 Iterationen.

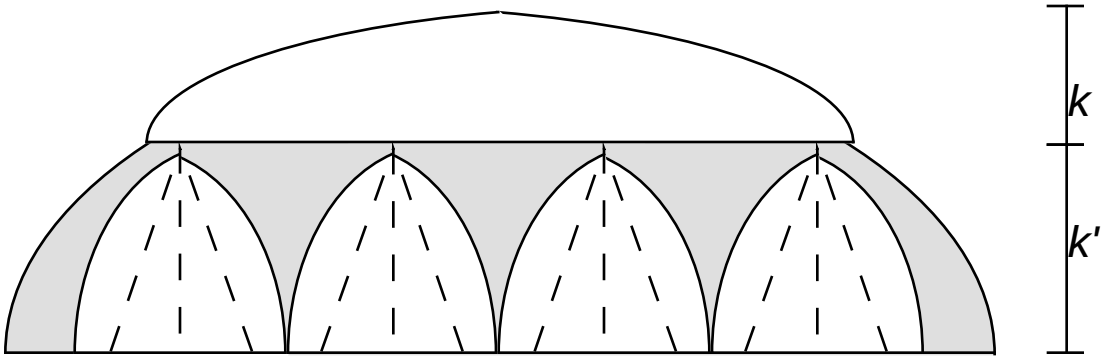

Abb. 5.8: Abarbeitung eines Suchbaums durch die k-Expansion mit modifizierter Expansionstiefe k' *in der tiefsten Suchbaumebene* (k' ≥ k)

(3) Bei der dritten Methode wird statt der letzten (ungünstigen) k-Expansion eine Reihe normaler 1-Expansionen durchgeführt. Die 1-Expansionen passen sich automatisch an die vorgegebene Tiefe des Suchbaums an, so daß kein Mehraufwand entsteht (Abb. 5.9). Die Anzahl von Teilbäumen ist für das obige Beispiel auf fünf gesunken. Zusätzlich kommen noch die 1-Expansionen dazu. Für vier Prozessoren entstehen dadurch insgesamt 13 Iterationen.

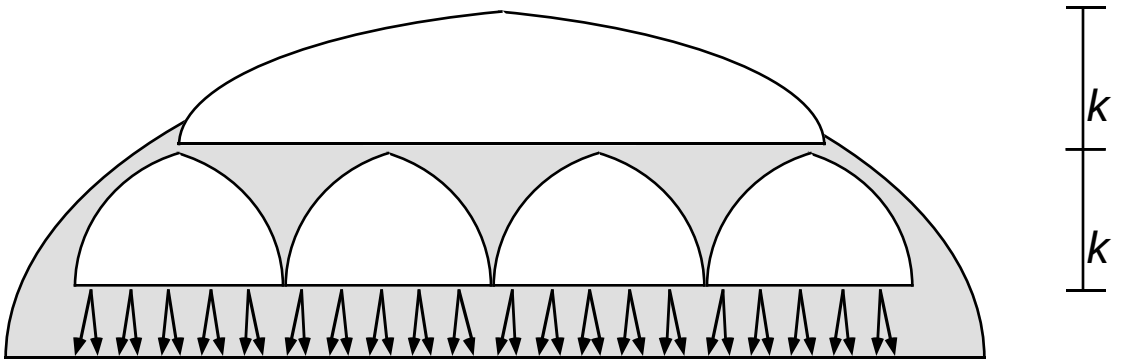

Abb. 5.9: Abarbeitung eines Suchbaums durch die k-Expansion mit fester Expansionstiefe k *und parallelen 1-Expansionen*

5.3.3 Selektive k-Expansion

Hier folgt eine weitere mögliche Ausprägung zur Realisierung der k-Expansion. Die *selektive* k-Expansion (SE) arbeitet in exakt der gleichen Weise wie die selektive Initialisierung aus Kapitel 3.2.3. Sie soll daher hier nur kurz wiederholt werden: Zu Beginn wird die Wurzel des zu expandierenden Teilbaums an alle Prozessoren versandt. Dann beschreitet jeder Prozessor einen einzelnen Pfad im Suchbaum. Um diesen Pfad abzugehen, führen die Prozessoren für eine bestimmte Zeit eine Reihe von Expansions- und Selektionsschritten durch. Dabei wird der Wurzelknoten expandiert und seine Nachfolger generiert. Statt alle Nachfolger in OPEN zu speichern, wird von jedem Prozessor nur ein Nachfolger weiterbearbeitet, so daß die vollständigen Pfade der Prozessoren verschieden sind (Abb. 3.4).

In dieser Ausprägung der k-Expansion wird die Tiefe *k* durch die Anzahl der heuristischen Beschneidungen im Suchbaum bestimmt. Die selektive k-Expansion endet, wenn jeder noch aktive Prozessor zumindest einen eigenen Knoten erhalten hat. Je mehr Zweige beschnitten werden, desto mehr Iterationen sind während der k-Expansion notwendig um zu terminieren. Nach der Terminierung hat eine große Anzahl von Prozessoren einen eindeutigen Knoten.

Für die Modellierung des Zeitaufwands T_{SE} der für eine B&B-Iteration mit direkter k-Expansion notwendig ist, werden die Parameter aus dem Kapitel 5.3.1 verwendet. (Die Generierungszeit T_{gen} ist für beide Methoden leicht unterschiedlich, aber diese Differenz ist in der Praxis zu vernachlässigen.) Neben dem logarithmischen Aufwand zur Berechnung des besten Knotens T_{red} und dem anschließenden Versenden dieses Knotens T_{send} sind in jeder Iteration ein Generierungs- und Evaluierungsschritt mit T_{gen} und T_{eval} notwendig. Bis in die dynamisch bestimmte Tiefe *k* werden alle *b* Nachfolger des aktuellen Knotens berechnet. Damit ergibt sich ein Zeitaufwand T_{SE} für eine B&B-Iteration mit selektiver k-Expansion von

$$T_{SE} = \log(P)T_{red} + T_{send} + k\,b\,(T_{gen} + T_{eval}) \qquad (5.9)$$

Vergleicht man die Zeitmodellierungen der zwei vorgestellten Ausprägungen, dann ergibt sich, daß die direkte k-Expansion mindestens so schnell durchgeführt werden kann wie die selektive Expansion. Das heißt, es gilt:

$$T_{DE} \leq T_{SE} \tag{5.10}$$

Die durch den B&B mit selektiver k-Expansion erreichbare asymptotische Beschleunigung ist die gleiche wie mit der direkten k-Expansion. Sie ist beeinflußt durch den logarithmischen Aufwand der Reduktionsschritte und ist gegeben durch

$$S_{SE}(P) = \frac{T(1)}{T_{SE}(P)} = O(\frac{P}{\log(P)}) \tag{5.11}$$

Auch hier gibt es neben der erreichbaren Beschleunigung noch weitere qualitative Kriterien. Die Ausprägung der selektiven k-Expansion für den parallelen B&B läßt sich wie folgt bewerten:

– Die Prozessoren leisten redundante Arbeit, da in jeder Expansion der Anfang der Pfade im Teilbaum von mehreren Prozessoren gemeinsam ausgewertet wird. Innerhalb einer k-Expansion der Tiefe k können mit P Prozessoren potentiell $k \cdot P$ Knoten expandiert werden. Bei einem Verzweigungsfaktor b des Suchbaums werden aber von der selektiven k-Expansion nur $N(b, k)$ Knoten ausgewertet.[3] Damit verhält sich die Auslastung asymptotisch proportional zu $\Theta(1/\log P)$, was auch in die Gleichung (5.11) einfließt.

– Wie auch bei der direkten k-Expansion ist keine Vollbeschäftigung der Prozessoren nach der Expansion garantiert. Sie ist aber auf Grund der dynamischen Beschneidung in vielen Fällen denkbar.

– Neben der üblichen OPEN-Menge ist noch eine zusätzliche Datenstruktur notwendig, um die Nachfolgerknoten während der Durchführung der selektiven k-Expansion zu speichern.

\+ Im Gegensatz zur direkten k-Expansion ist die selektive k-Expansion auf allgemeine Suchbäume anwendbar, da nur die generische Expansionsfunktion verwendet wird.

\+ Eine Beschneidung von ungünstigen Knoten ist auch innerhalb der k-Expansion möglich. Damit können am Ende der k-Expansion viele Prozessoren mit unbeschnittenen Knoten versorgt werden. Das heißt, die Effektivität der k-Expansion als Lastverteilung ist höher als bei der direkten k-Expansion.

---

[3] Die Definition von $N(b, d)$ ist in Gleichung (5.1) gegeben.

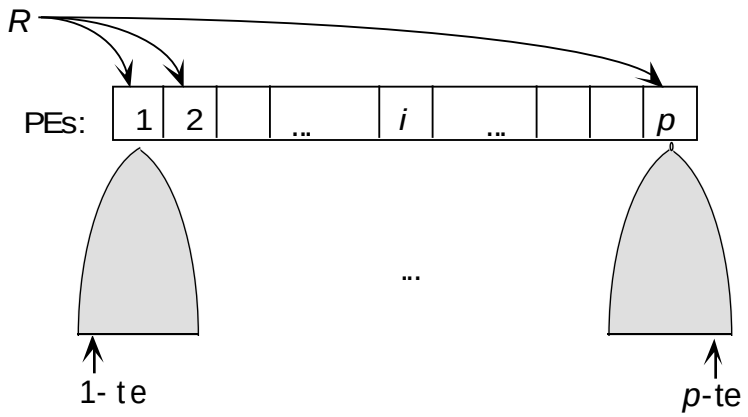

Abb. 5.10: Einfache Ausführung einer k-Expansion mit P *Prozessoren (PEs)*

5.4 Variationen

Nach der Beschreibung des Grundkonzepts in Kapitel 5.2 und den möglichen Realisierungen in Kapitel 5.3, sollen hier die unterschiedlichen Varianten zur Ausführung der k-Expansion besprochen werden. Bei der jeweiligen Ausführungsart ist die Realisierung der k-Expansion nicht festgelegt. Vier Varianten der Ausführung können unterschieden werden: die einfache, die überlappende, die mehrfache und die verwobene k-Expansion. Sie werden im folgenden kurz dargestellt.

5.4.1 Einfache k-Expansion

Die *einfache* k-Expansion bearbeitet in einem Iterationsschritt genau einen Teilbaum der Tiefe *k* mit mehreren Prozessoren (Abb. 5.10). Diese Variante entspricht der im Grundkonzept beschriebenen Ausführung. Das heißt, eine globale Wurzel *R* wird an alle *P* Prozessoren versandt.

Neben den schon in Kapitel 5.2.3 aufgeführten Punkten lassen sich speziell in Hinsicht auf den Vergleich mit den anderen Varianten weitere Bewertungen hinzufügen:

± Einerseits führt die einfache k-Expansion eine globale Bestensuche auf Teilbaumebene durch. Andererseits werden alle Prozessoren in die gleiche Sackgasse hineingeführt. Das heißt, in möglicherweise ungünstigen Teilbäumen wird viel Suchaufwand verbraucht.

– Zur Auswahl des Wurzelknotens für die nächste Iteration ist globale Kommunikation notwendig. Der Aufwand dieser Kommunikation ist von der Anzahl der Prozessoren abhängig.

– Die Größe der in einer Iteration expandierten Teilbäume ist durch die Prozessoranzahl festgelegt.

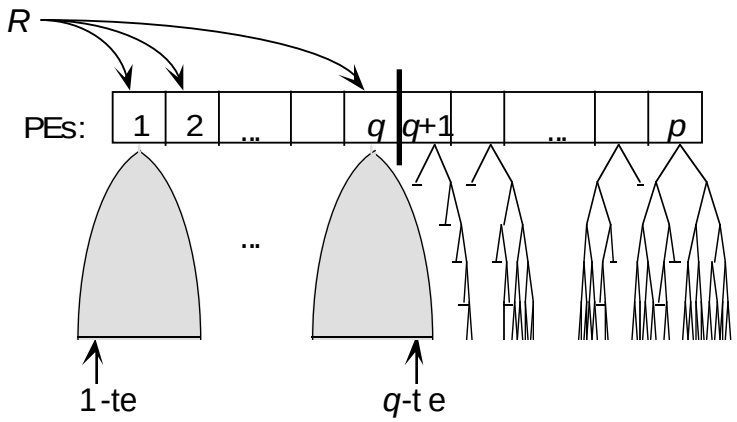

Abb. 5.11 Überlappende Ausführung von einer k-Expansion (Prozessor 1 bis q*) und mehreren 1-Expansionen (Prozessor* q *+ 1 bis* P*)*

– Bei ungünstigem Verhältnis von Teilbaumtiefe *k* zu Suchbaumtiefe *d* kann eine starke Sequentialisierung der Suche auftreten (Kapitel 5.3.2). Bei der selektiven k-Expansion fällt der vorige Nachteil zwar nicht ins Gewicht, dafür sind aber redundante Berechnungen notwendig.

5.4.2 Überlappende k-Expansion

Die nächste Variation der Ausführung stellt die *überlappende* k-Expansion dar. Wie bei der einfachen k-Expansion wird nur ein Wurzelknoten *R* durch eine Reduktion bestimmt. Statt nun diese Wurzel auf alle *P* Prozessoren durch ein Broadcast zu verteilen, wird sie nur an $q < P$ Prozessoren versandt. Diese *q* Prozessoren führen dann eine einfache k-Expansion durch (Kapitel 5.3.1). Die restlichen $P - q$ Prozessoren fahren mit der üblichen 1-Expansion fort. Je nach Realisierung der k-Expansion (direkt bzw. selektiv) werden eine bzw. mehrere 1-Expansionen durchgeführt. Dem üblichen parallelen B&B kann dadurch eine k-Expansion überlagert werden (Abb. 5.11).

Die überlappende k-Expansion hat im Vergleich zu den anderen Variationen eine besondere Eigenschaft. Durch die gezielte Auswahl der beteiligten Prozessoren, kann die überlappende k-Expansion als expliziter Lastverteilungsmechanismus eingesetzt werden. Wenn der Trigger des normalen B&B mit 1-Expansionen feuert, dann wird ein geeigneter Wurzelknoten ausgewählt und an alle unbeschäftigten Prozessoren versandt. Durch die überlappende k-Expansion werden je nach Realisierung und Beschaffenheit des Suchbaums die Prozessoren wieder mit Knoten versorgt.

5.4.3 Mehrfache k-Expansion

Bei der *mehrfachen* k-Expansion wird die einfache k-Expansion so erweitert, daß mehrere Teilbäume gleichzeitig bearbeitet werden. Statt nur einer einzelnen Teilbaumwurzel wird nun eine Reihe von Wurzelknoten R_1 bis R_x ausgewählt

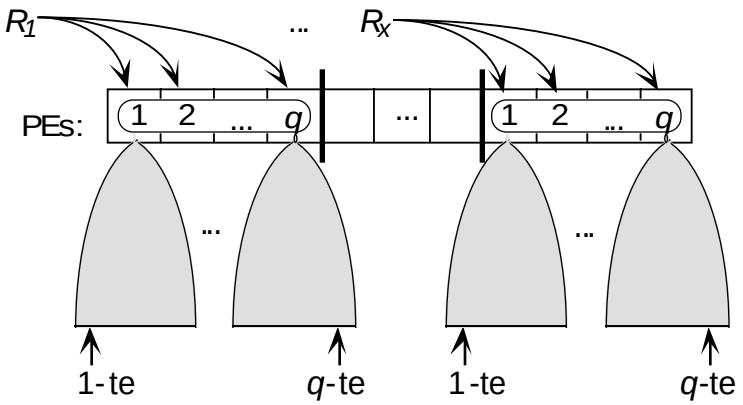

Abb. 5.12 Mehrfache Ausführung einer k-Expansion mit jeweils q *Prozessoren*

und mit jeweils q Prozessoren (bei $P = x \cdot q$) bearbeitet. Die zu den Wurzeln gehörenden Teilbäume werden je nach Ausprägung der k-Expansion auf die gleiche Art expandiert wie bei der einfachen k-Expansion (Abb. 5.12). Die Tiefe der Teilbäume richtet sich nach der Anzahl x der gleichzeitig expandierten Teilbäume und der gewählten Ausprägung. Bei der direkten k-Expansion sollte die Tiefe so angepaßt werden, daß möglichst viele Prozessoren beschäftigt werden. Die Wahl von k wurde schon in Kapitel 5.3.2 diskutiert.

Mit der mehrfach ausgeführten k-Expansion können einige Nachteile der einfachen k-Expansion vermieden werden:

+ Es ist keine globale Kommunikation notwendig, da nur für jeweils q Prozessoren der günstigste lokale Wurzelknoten ausgewählt werden muß.
+ Die Größe bzw. die Anzahl der gleichzeitig expandierten Teilbäume ist frei wählbar. Damit kann die mehrfache k-Expansion angepaßt werden, um z.B. eine zu starke Sequentialisierung zu vermeiden.
+ Außerdem werden nur konstant viele Prozessoren in einen evtl. ungünstigen Teilbaum hineingezogen. Damit wird eine übermäßige Verschwendung der Resourcen vermieden.
± Die globale Suchstrategie wird auf Gruppen der Größe q beschränkt. Im Vergleich zu dem üblichen B&B mit 1-Expansion werden statt P günstigste Knoten nur die x günstigsten Knoten ausgewählt, mit $x \ll P$. Das heißt, es handelt sich um eine semi-globale Suchstrategie.
– Da andererseits mehrere Wurzelknoten ausgewählt und an die jeweils beteiligten Prozessoren versandt werden, ist die bisherige Verwendung der Busverbindung nicht mehr möglich.

5.4.4 Verwobene k-Expansion

Die *verwobene* k-Expansion entsteht durch drei Modifikationen der mehrfachen k-Expansion. Zum einen werden die Prozessorgruppen, die jeweils eine k-Expansion durchführen, ineinander geschoben, so daß mit jedem Prozessor eine Gruppe beginnt (Abb. 5.13). Da sich die Gruppen überlagern,

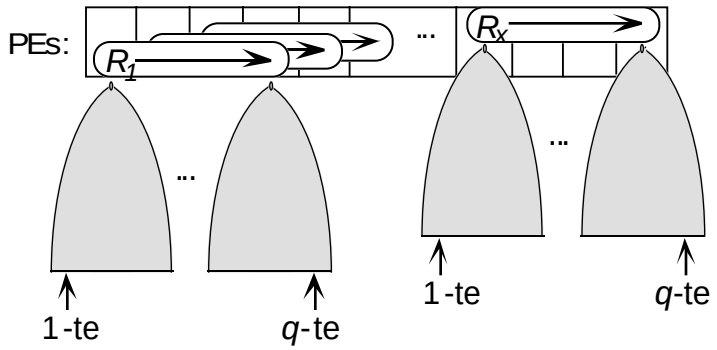

Abb. 5.13: Verwobene Ausführung von mehreren k-Expansionen mit jeweils q *(überlagerten) Prozessoren*

nimmt jeder Prozessor sukzessive an mehreren k-Expansionen teil. Als zweite Modifikation wird der lokale Wurzelknoten der Teilbäume einer k-Expansion nicht mehr aus allen Prozessoren der zugehörigen Gruppe ausgewählt, sondern nur von dem ersten Prozessor der Gruppe. Falls ein Prozessor keinen Knoten zur Verfügung stellen kann, führt die zugehörige Gruppe diese k-Expansion nicht durch. Die dritte Modifikation ist die alternierende Ausführung von Knotengenerierung und Verschieben des Wurzelknotens. Insgesamt entspricht die verwobene k-Expansion einem Verteilen der Nachfolger aus Tiefe *k* an die Nachbarn.

Einige der bei der mehrfachen k-Expansion verbleibenden Nachteile können durch die verwobene k-Expansion beseitigt werden:

+ Die Auswahl des lokalen Wurzelknotens geschieht strikt lokal. Damit entfällt die semi-globale Kommunikation der mehrfachen k-Expansion.
+ Das Versenden des lokalen Wurzelknotens wird mit der Knotengenerierung alternierend durchgeführt. Daher muß die Wurzel nur lokal an die direkten Nachbarn verschoben werden, d.h. auch für diesen Schritt entfällt die globale Kommunikation.
+ Trotz Überlagerung mehrerer k-Expansionen muß pro Prozessor nur jeweils eine lokale Wurzel gespeichert werden, da die Wurzel nach der Expansion weitergeschoben wird.
± Es kann keine globale Suchstrategie mehr verfolgt werden.

Auf unterschiedlichen Topologien sind verschiedene Implementierungen der verwobenen k-Expansion möglich. Auf einem Ring kann der obige Algorithmus direkt übertragen werden. In dem hier verwendeten 2-dimensionalen Torus alterniert die Transferrichtung einer vollständigen k-Expansion in den beiden Dimensionen. Das heißt, für eine k-Expansion werden die lokalen Wurzel z.B. nach Westen und für die nächste k-Expansion nach Süden geschoben.

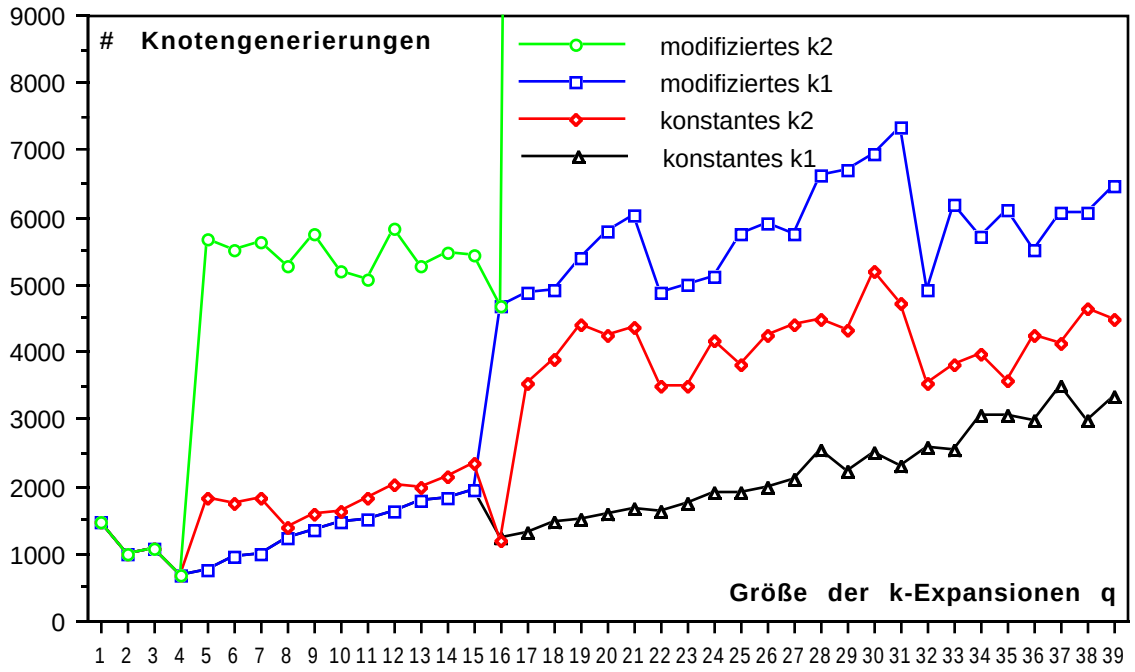

Abb. 5.15: Auswirkung der verschiedene Möglichkeiten zur Wahl von k *auf die Anzahl von generierten Knoten abhängig von der Größe* q *der verwobenen k-Expansion am Beispiel von Benchmark* `bm16`

5.5 Experimentelle Ergebnisse

Von den vorgestellten Ansätzen der k-Expansion ist die verwobene k-Expansion am vielversprechendsten. Sie soll daher in einigen Experimenten weiter untersucht werden. Zunächst gilt es die Anzahl der an einer k-Expansion beteiligten Prozessoren zu bestimmen. In Abb. 5.15 wird die Auswirkung der Wahl von k auf den B&B mit allgemeiner, verwobener k-Expansion näher beleuchtet. Dazu wurde das Benchmark-Problem `bm16` beispielhaft mit verschiedenen Prozessoranzahlen q für eine k-Expansion durchgerechnet (q = 1, …, 39).[4] Außerdem wurde der Parameter k nach den Möglichkeiten aus Kapitel 5.3.2 gewählt (k_1 bzw. k_2 und konstantes bzw. modifiziertes k). Aufgetragen ist für jede der vier Kombinationen die Anzahl paralleler Knotengenerierungen, als abstraktes und vergleichbares Zeitmaß.

In beiden Kombinationen erbringen das konstante k bzw. die Wahl von k_1 die günstigeren Ergebnisse. Daraus folgt, daß die kleineren zu expandierenden Teilbäume auf Grund der heuristischen Beschneidung wichtiger sind, als die daraus resultierende Lastverteilung. Jedoch ist ein Mindestmaß an Lastverteilung

[4] Dabei wurde die selektive Initialisierung und die statische Beschneidung verwendet. Außerdem ist zum Ausgleich von eventuellen Symmetrien des Suchbaums die Zuordnung der Nachfolger auf die Prozessoren randomisiert worden.

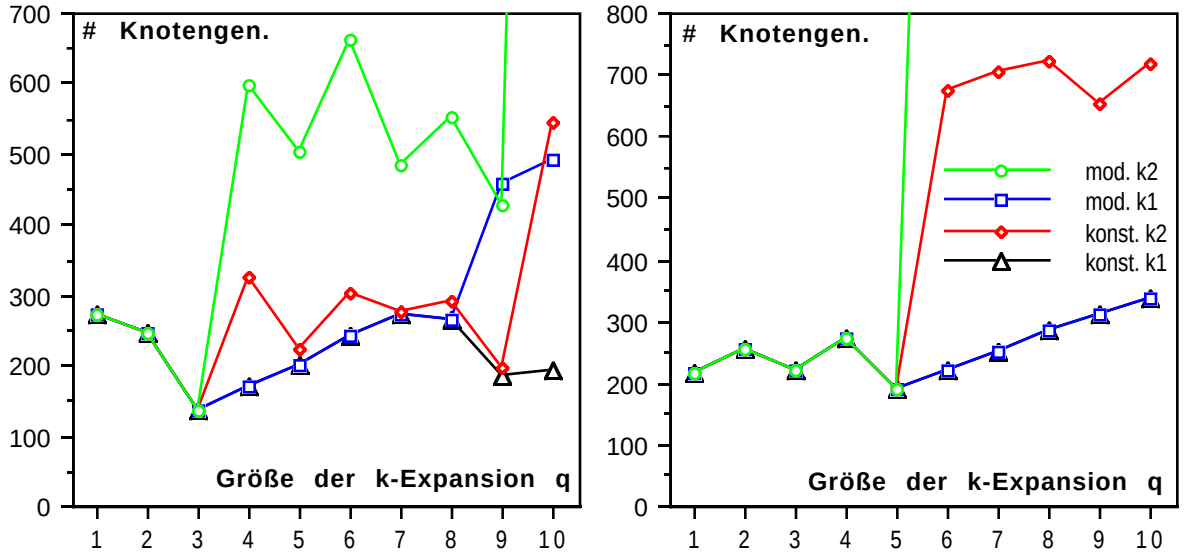

Abb. 5.16: Weitere Beispiele mit unterschiedlichem Verzweigungsfaktor zur Wahl von k *(links:* `bm8` *mit* b = 3*, rechts:* `bm23` *mit* b = 5*)*

vonnöten, so daß schlußendlich die Größe $q = 4$ der k-Expansion sich für diese Probleminstanz am günstigsten erweist. Für die zwei erfaßten Potenzen (4 und 16) des Verzweigungsfaktors $b = 4$ fallen die jeweiligen Ergebnisse von k_1 und k_2 aufeinander, da dort $k_1 = k_2$ gilt. Dennoch entstehen bei $q = 16$ zwei unterschiedliche Ergebnisse, da dort $d \bmod k \neq 0$ mit $d = 15$ gilt und dies für ein konstantes bzw. modifiziertes k unterschiedlich aufgelöst wird.

In Abbildung 5.16 sind noch zwei weitere Beispiele mit unterschiedlichem Verzweigungsfaktor gegeben. Links ist das Problem `bm8` mit $b = 3$ und rechts das Problem `bm23` mit $b = 5$ dargestellt. Für beide Beispiele treffen dieselben Aussagen zu wie oben, nur daß sich durch den veränderten Verzweigungsfaktor die optimale Größe der k-Expansion verändert.

Ein weiterer Aspekt der Untersuchungen ist das Laufzeitverhalten der k-Expansion. Dazu wird einerseits die Laufzeit direkt gemessen und zum anderen die Anzahl von parallelen Knotenexpansionen als abstraktes Zeitmaß betrachtet. Die Prozessoranzahl q pro k-Expansion wurde auf den Verzweigungsfaktor b gesetzt. Außerdem wurde nur eine statische Beschneidung der Knoten vorgenommen. Die mit der k-Expansion lösbaren Benchmarks und die dazugehörige Laufzeit bzw. Anzahl von Knotengenerationen ist in Abb. 5.17 gegeben.

Von den 30 Benchmark-Problemen konnte nur ein kleiner Teil mit der hier beschriebenen Version der verwobenen k-Expansion gelöst werden. Trotz der speichereffizienten Tiefensuche kommt es bei einigen Problemen zum Speicherüberlauf. Dies ist überraschend, da bei dem Flüssigkeitsmodell aus Kapitel 4 mit derselben Auswahlstrategie alle Benchmark-Probleme gelöst werden konnten. Der Grund liegt in der Berücksichtigung von Prozessoren mit vollständig gefüllter OPEN-Menge. Bei dem Flüssigkeitsmodell werden nicht nur die

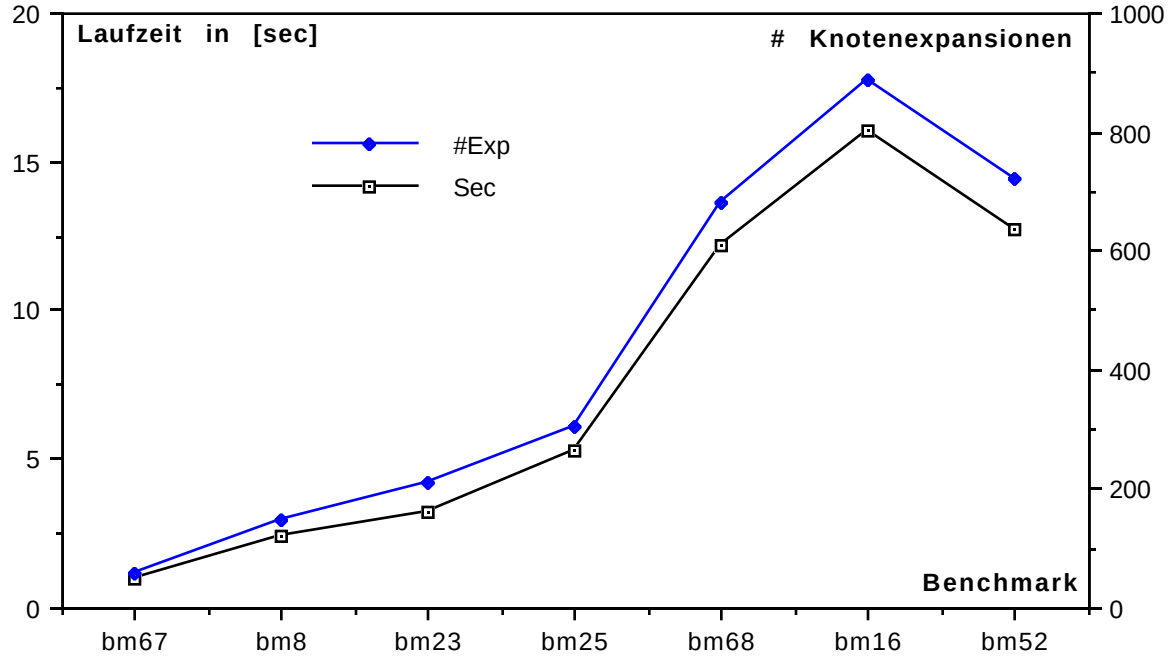

Abb. 5.17: Auswertung der lösbaren Benchmark-Probleme mit der verwobenen k-Expansion

wartenden Prozessoren wieder mit Arbeit versorgt, sondern auch die überbeschäftigten Prozessoren durch Verschieben der Lastelemente entlastet. Dagegen kann die k-Expansion die überbeschäftigten Prozessoren nicht berücksichtigen. Selbst ein "Ausschalten" dieser Prozessoren ist nicht möglich, da ansonsten das Schema der verwobenen k-Expansion in der Form nicht mehr durchgeführt werden kann. Das heißt, die k-Expansion erreicht bei ungünstiger Beschneidung viel früher die Grenzen der Speicherkapazität der einzelnen Prozessoren als das Flüssigkeitsmodell.

Abschließend soll untersucht werden, wie sich die Struktur des Suchbaum in zunehmender Tiefe entwickelt. Dies ist für die k-Expansion insofern wichtig, als daß ihre Effektivität bei der Lastverteilung von der Struktur abhängig ist. Die Struktur des Suchbaums kann bei konstanten Verzweigungsfaktor b durch den Grad der Beschneidung einfach charakterisiert werden. Der *Beschneidungsgrad* w_k gibt für die Tiefe k den prozentualen Anteil der beschnittenen Knoten relativ zu der Gesamtzahl von Knoten an. Ein Knoten wird genau dann beschnitten, wenn seine Kosten die der derzeit besten Lösung (obere Schranke) überschreiten. In Abb. 5.18 ist der Beschneidungsgrad der Suchbäume für die Benchmark-Probleme aus Kapitel 7.1 angegeben. Dabei wurde über die Probleme mit gleichem Verzweigungsfaktor gemittelt. Als statische obere Schranke wurde die erste heuristische Lösung gewählt.

Mit zunehmender Tiefe steigt für alle Probleme der Anteil von beschnittenen Knoten an. Je größer der Verzweigungsfaktor ist, desto früher steigt der Grad der Beschneidung. Für die hier verwendete Heuristik zur

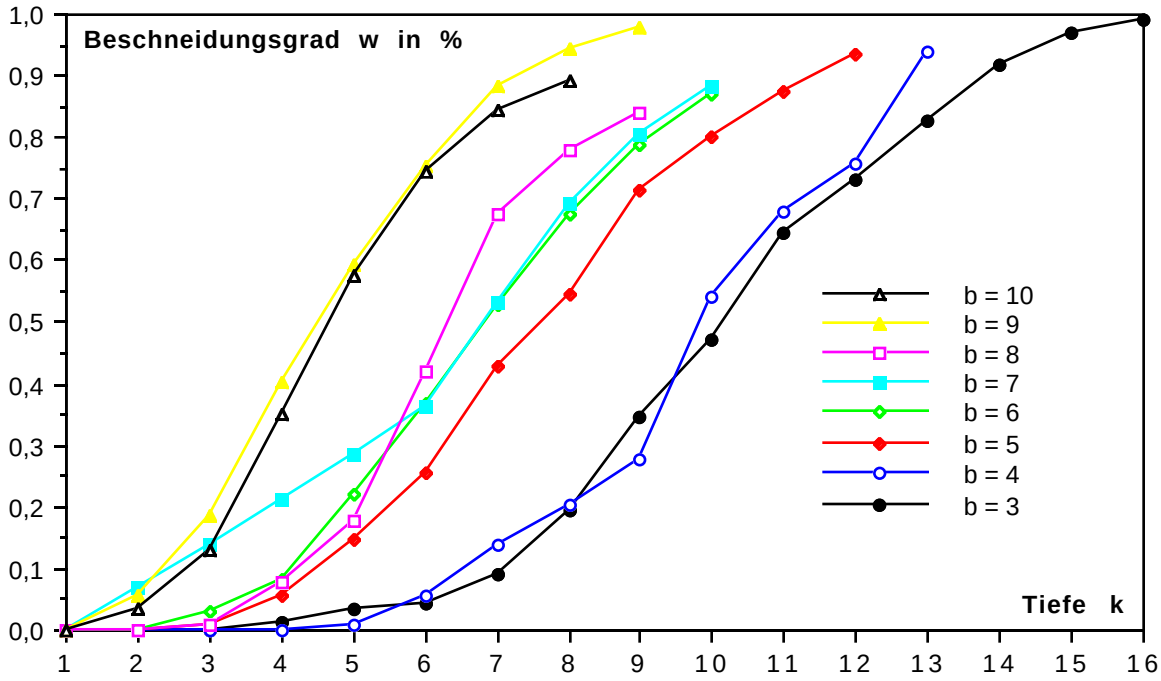

Abb. 5.18: Über den Benchmark-Satz gemittelte prozentuale Beschneidung des Suchbaums abhängig von der Baumtiefe k *für verschiedene Verzweigungsfaktoren* b

Knotenbewertung (Kapitel 7.2) bedeutet dies, daß sie in immer größeren Tiefen die tatsächlichen Kosten besser abschätzen kann. Daraus folgt für die k-Expansion, daß sie mit zunehmender Tiefe ineffektiver wird, d.h. sie kann immer weniger Prozessoren mit Knoten versorgen. Die verwendete Anwendungsdomäne ist für die k-Expansion weniger geeignet. Dagegen ist die k-Expansion sicherlich für Domänen mit gleichbleibend wirksamer Heuristik effektiver.

6. Zusammenfassung

In dieser Arbeit wird die Parallelisierung des allgemeinen und exakten Lösungsverfahren Branch-and-bound (B&B) auf massiv-parallelen Rechnerstrukturen betrachtet. Die angegangene Problemstellung ist die sowohl effiziente als auch skalierbare Lastverteilung für feinkörniges B&B. Dabei werden als Ziele die Wiederbeschäftigung möglichst vieler stillstehender Prozessoren (Load-sharing) bzw. eine Gleichverteilung der Last über alle Prozessoren (Load-balancing) angestrebt. Die parallelen Baumsuchverfahren wie B&B bereiten besondere Schwierigkeiten, da die Verteilung der Last nicht vorhersehbar und stark dynamisch ist. Als exemplarische, NP-harte Anwendungsdomäne wurden statische, non-operationale Belegungsplanungsprobleme ohne Reihenfolgebedingungen eingesetzt.

6.1 Ergebnisse

Statische Lastverteilung (Initialisierung):

Die statische Lastverteilung wird vor dem eigentlichen B&B-Algorithmus ausgeführt. Sie strebt eine Versorgung aller Prozessoren mit Arbeit durch ein anfängliches Expandieren und Abarbeiten des Suchbaums an. Es wurde eine Reihe von Verfahren vorgestellt und auf ihre Eignung hin verglichen.

- Von allen betrachteten Initialisierungen sind drei Verfahren (selektive, direkte und erweiterte direkte Initialisierung) ähnlich günstig. Sie unterscheiden sich zwar in ihrer Effektivität bzgl. Load-sharing, aber die Effizienz des Gesamtalgorithmus wird dadurch wenig beeinträchtigt.
- Die übliche Wurzelinitialisierung ist dagegen für die Baumsuche mit lokaler, dynamischer Lastverteilung auf massiv-parallelen Rechnern mit größerem Netzwerkdurchmesser nicht sinnvoll einsetzbar.
- Eine Abhängigkeit der Effizienz der Initialisierung von der Problemgröße konnte bei keinem Verfahren festgestellt werden.

- Die statische Lastverteilung wirkt sich überproportional stark auf die Laufzeit des nachfolgenden Branch-and-bound-Algorithmus aus. Dies stellt die bisher unterschätzte Bedeutung der statischen Lastverteilung für reale Problemstellungen heraus.

Dynamische Lastverteilung (Flüssigkeitsmodell): 

Die dynamische Lastverteilung arbeitet nebenläufig zu dem Anwendungsalgorithmus und versucht die dort entstehenden Unausgewogenheiten in der Verteilung der Arbeit wieder auszugleichen. Dazu wurde ein vereinfachtes, gut skalierbares Flüssigkeitsmodell als erste synchrone, lokale Lastverteilung entwickelt. Die Methode wurde mit der bekannten, aus dem Asynchronen übertragenen Nachbarschafts-Mittelung verglichen. In diesem Zusammenhang wurde auch der periodische Trigger-Mechanismus untersucht.

- Das Flüssigkeitsmodell leistet eine Kombination von Load-balancing und Load-sharing und ist damit für sehr dynamische Anwendungsprozesse gut geeignet.
- Zum analytischen Vergleich wurde als ein realistischeres Aufwandsmaß die die Menge der übertragenen Information statt der üblichen Anzahl von Kommunikationsschritten verwendet.
- Der in der Prozessoranzahl für mehrdimensionale Tori bisher quadratische Zeitaufwand wird durch das Flüssigkeitsmodell auf einen linearen Aufwand reduziert.
- Das Flüssigkeitsmodell ist auch bzgl. den Konstanten signifikant effizienter. Es konnte eine Effizienzverbesserung um 30 % für den parallelen Branch-and-bound-Algorithmus erreicht werden.

Implizite Lastverteilung (k-Expansion): 

Die implizite Lastverteilung ist eine Verschmelzung des B&B mit dem Lastverteilungsprozeß, so daß durch die veränderte Bearbeitung des Suchbaums die Last automatisch auf die Prozessoren verteilt wird. Dazu wurde das neuartige Konzept der k-Expansion eingeführt.

- Für das Konzept der k-Expansion gibt es eine Vielzahl von Variationsmöglichkeiten mit unterschiedlichen Eigenschaften.
- Der B&B mit k-Expansion unterstützt eine automatische Lastverteilung, falls der effektive Verzweigungsfaktor hinreichend groß ist.
- Mit der k-Expansion kann eine globale Suchstrategie approximiert werden.
- Der aufgrund knapper Resourcen auftretende Speicherüberlauf in den einzelnen Prozessoren kann nicht so einfach bewältigt werden wie im Flüssigkeitsmodell.

Zur Validierung der Ergebnisse wurden Simulationen und Experimente mit einem Satz von 30 Benchmark-Problemen durchgeführt. Die zugrunde liegende SIMD-Architektur war der feinkörnige Rechner MasPar MP-1 mit 16.384

Prozessoren in einem 2-dimensionalen Torus. Als exemplarische, NP-harte Anwendungsdomäne wurden statische, non-operationale Belegungsplanungsprobleme mit identischen, parallelen Maschinen betrachtet.

Zusammenfassend kann man sagen, daß die massiv-parallele Ausführung der stark dynamischen Baumsuch-Algorithmen auf feinkörnigen SIMD-Architekturen bei entsprechender Initialisierung der Prozessoren auch mit strikt lokalen Lastverteilungsmethoden effizient möglich ist.

6.2 Ausblick

Weitere Forschungsarbeiten, die im Zusammenhang mit der hier betrachteten diskreten parallelen Optimierung durch Branch-and-bound (B&B) stehen, können folgende vier Aspekte betreffen.

Neben der Berechnung der optimalen Lösung, wie sie in dieser Arbeit vorgenommen wurde, ist eine einfache Erweiterung des verwendeten Algorithmus die Suche nach suboptimalen Lösungen. In [Li84a] wird ein paralleler B&B-Algorithmus beschrieben, der suboptimale Lösungen mit Kosten innerhalb eines vorgegebenen Abstands zum nicht bekannten Optimum berechnet (ε-Approximation). Nach [Wah82b] kann durch eine lineare Reduktion der geforderten Genauigkeit der Lösung eine exponentielle Reduktion der mittleren Laufzeit des B&B für NP-harte Probleme erreicht werden. Offen ist die Frage, wie der approximative parallele B&B im Vergleich mit anderen suboptimalen parallelen Verfahren wie z.B. Genetische Algorithmen, abschneidet.

Der Ansatz des approximativen B&B kann direkt auf eine weitere Anforderung angewendet werden. Bei dieser zusätzlichen Randbedingung handelt es sich um die Realzeitfähigkeit des B&B. In [Wah82b] wird folgende Methode vorgeschlagen: Mit sukzessiver Halbierung der verbleibenden Rechenzeit wird die geforderte Lösungsqualität auch "halbiert" bzw. das Lösungsintervall verdoppelt. Dadurch kann innerhalb einer vorgegebenen Zeitschranke eine approximative Lösung mit bekannter Qualität berechnet werden, vorausgesetzt es kann eine heuristische Lösung vor Ausführung des B&B gefunden werden. Da die Anforderung der Realzeitfähigkeit des B&B unabhängig von der sequentiellen oder parallelen Ausführung ist, wurde sie hier nicht behandelt.

Ein weiterer, in dieser Arbeit bewußt ausgeklammerter Aspekt ist die Speicherkapazität. Wie auch schon bei sequentiellen Rechnern tritt bei den Parallelrechnern verstärkt das Problem des begrenzten Speicherplatzes auf. Es gibt einige Möglichkeiten mit diesem Problem umzugehen. Zum einen sind dies von vornherein speicherbeschränkte Algorithmen. Zum anderen ist eine Verzögerung und teilweise Vermeidung des Speicherproblems durch Umverteilen der Speicherelemente möglich. Dadurch wird erreicht, daß der

vorhandene Speicher zur Beschleunigung der Suche besser ausgenutzt wird. Diese Umverteilung der Speicherelemente kann in Lastverteilungsalgorithmen eingebunden werden.

Ein weiterführender Aspekt bei den hier betrachteten massiv-parallelen B&B-Varianten ist ihre die Anwendung auf reale Problemstellungen. Zwar ändern realistischere Anwendungen nichts an den hier gewonnenen Aussagen, aber durch zusätzliche Randbedingungen kann sicherlich eine weitere Beschleunigung gewonnen werden. Zusätzlich werden bei umfangreicheren Problemgrößen die Grenzen der Anwendbarkeit des massiv-parallelen B&B aufgezeigt.

7. Anhänge

7.1 Benchmark-Probleme

Nach [Schneider91] ist ein *Benchmark* definiert "als (1) allgemein Bezugspunkt, Fixpunkt, Merkpunkt, von dem aus Messungen gemacht werden und (2) ein Test zum Vergleich verschiedener Geräte". Hier wird unter einem Benchmark eine Menge von Problemen verstanden, welche zu einem fairen Vergleich und Bewertung von Algorithmen geeignet sind.

Für die Belegungsplanung mit parallelen Maschinen gibt es eine Reihe von Benchmarks. In [Baker74a] sind 16 Probleme mit Lösungen für acht Jobs auf einer Maschine zur Minimierung der mittleren Verspätung angegeben ($1 / d_j / \sum T$).[1] Leider existieren für diese Probleme keine experimentellen Ergebnisse. In [Baker74b] wird für eine Maschine die maximale Verspätung mit Freigabezeiten minimiert ($1 / r_j$, prec$/ T_{max}$). Für n Jobs mit $n = 10, 20, 30$ werden 93 % der Probleme gelöst. In [Barnes77] werden 15 Probleme zur Minimierung der gewichteten Verspätung für parallele, identische Maschinen beschrieben. Die Fälligkeitstermine sind identisch mit den Bearbeitungszeiten ($P / d_j = t_j / \sum w_j L_j$). Neben der optimalen Lösung und dem Zeitaufwand werden noch die Anzahl der benötigten Iterationen zum Finden des Optimums und zum Beweisen des Optimums angegeben. In [Elmaghraby74] wird die mittlere Verspätung auf identischen Maschinen mit Fälligkeitsterminen identisch mit den Bearbeitungszeiten behandelt ($P / d_j = t_j / \sum w_j L_j$). Experimentelle Ergebnisse werden für das erste Problem aus [Rothkopf66] angegeben. Für die weiteren Ergebnisse fehlt die Angabe der Problemstellungen.

Leider wurde für den hier betrachteten einfachsten Problemtyp im non-operationalen Belegungsplanung *(1/ /L_{max})* in der Literatur kein Benchmark gefunden. Daher wird im folgenden ein erster Benchmark für diesen Problemtyp eingeführt. Die Probleme dieses Benchmarks sind systematisch erzeugt. Die

[1] In Klammer ist zur Vollständigkeit die Klassifikation des Problem nach [Lawler89] angegeben.

N()	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	***b*** →
1	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
2	0	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	
3	1	1	1	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3	3	3	4	4	4	4	
4	1	2	2	2	3	3	3	3	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	
5	1	2	3	3	3	4	4	4	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6	7	
6	2	3	3	4	4	5	5	5	6	6	6	6	6	7	7	7	7	7	7	7	8	8	8	8	
7	2	3	4	4	5	5	6	6	7	7	7	7	8	8	8	8	8	8	9	9	9	9	9	9	
8	2	3	4	5	6	6	7	7	8	8	8	8	9	9	9	9	10	10	10	10	10	10	11	11	
9	3	4	5	6	7	7	8	8	9	9	9	10	10	10	10	11	11	11	11	11	12	12	12	12	
10	3	4	6	7	7	8	9	9	10	10	10	11	11	11	12	12	12	12	13	13	13	13	13	13	
11	3	5	6	7	8	9	9	10	11	11	11	12	12	12	13	13	13	14	14	14	14	14	15	15	
12	3	5	7	8	9	10	10	11	12	12	12	13	13	14	14	14	15	15	15	15	16	16	16	16	
13	4	6	7	9	10	11	11	12	13	13	14	14	14	15	15	16	16	16	16	17	17	17	17	18	
14	4	6	8	9	10	11	12	13	14	14	15	15	16	16	16	17	17	17	18	18	18	19	19	19	
15	4	7	9	10	11	12	13	14	15	15	16	16	17	17	18	18	18	19	19	19	20	20	20	20	
16	5	7	9	11	12	13	14	15	16	16	17	17	18	18	19	19	20	20	20	21	21	21	22	22	
17	5	8	10	11	13	14	15	16	17	17	18	18	19	20	20	20	21	21	22	22	22	23	23	23	
18	5	8	10	12	14	15	16	17	18	18	19	20	20	21	21	22	22	23	23	23	24	24	24	25	
19	6	9	11	13	14	16	17	18	19	19	20	21	21	22	22	23	23	24	24	25	25	25	26	26	
20	6	9	12	14	15	16	18	19	20	20	21	22	22	23	24	24	25	25	26	26	26	27	27	27	
21	6	10	12	14	16	17	19	20	21	21	22	23	24	24	25	25	26	26	27	27	28	28	29	29	
22	6	10	13	15	17	18	19	21	22	22	23	24	25	25	26	27	27	28	28	29	29	29	30	30	
23	7	11	13	16	17	19	20	21	23	23	24	25	26	27	27	28	28	29	29	30	30	31	31	32	
24	7	11	14	16	18	20	21	22	24	25	25	26	27	28	28	29	30	30	31	31	32	32	33	33	
25	7	12	15	17	19	21	22	23	25	26	27	27	28	29	30	30	31	31	32	33	33	34	34	34	
26	8	12	15	18	20	22	23	24	26	27	28	28	29	30	31	32	32	33	33	34	34	35	35	36	
27	8	13	16	18	21	22	24	25	27	28	29	30	30	31	32	33	33	34	35	35	36	36	37	37	
28	8	13	16	19	21	23	25	26	28	29	30	31	32	32	33	34	35	35	36	37	37	38	38	39	
29	9	14	17	20	22	24	26	27	29	30	31	32	33	34	34	35	36	37	37	38	38	39	40	40	
30	9	14	18	21	23	25	27	28	30	31	32	33	34	35	36	36	37	38	39	39	40	40	41	41	
d ↓																									

Abb. 7.1: Größe der unbeschnittenen Suchbäume abhängig vom Verzweigungsfaktor b *und der Tiefe* d *in Zehnerpotenzen*

Systematik berücksichtigt die zwei wichtigsten Kennzahlen von Scheduling-Problemen: die Anzahl der Maschinen *m* und der Jobs *n*. In der Darstellung des Problems als Suchbaum entsprechen diese Kennzahlen dem Verzweigungsfaktor *b* und der maximale Tiefe *d* des Suchbaums.

In Abb. 7.1 sind für unterschiedliche Kombinationen von *b* und *d* die Größen der vollständigen (unbeschnittenen) Suchbäumen berechnet. Die Größe ist dabei als Exponent der Zehnerpotenz angegeben. Die Suchbaumgröße bildet eine erste Abschätzung der Komplexität eines diskreten Optimierungsproblems. Die Suchbaumgröße *N* wird mit der Anzahl von Knoten des Baums, abhängig von dem Verzweigungsgrad *b* und der Baumtiefe *d* angegeben. Bei konstantem $b > 1$ ergibt sich aus der geometrischen Reihe für *N*:

$$N(b, d) := \frac{b^{d+1} - 1}{b - 1} \tag{7.1}$$

Diese Systematik in der Erzeugung der Probleme legt keineswegs den Schwierigkeitsgrad des Problems endgültig fest. Denn zu jedem Paar von

Name	m	n	t_i	$W/\ 10^5$	Klasse
bm8	3	15	8, 7, 5, 2, 4, 6, 4, 8, 7, 5, 6, 4, 5, 7, 4	1,44	einfach
bm16	4	15	4, 6, 7, 8, 3, 4, 5, 4, 3, 5, 6, 4, 6, 5, 3	11,21	mittel
bm23	5	13	4, 5, 6, 8, 9, 5, 6, 7, 3, 4, 6, 5, 3	2,15	einfach
bm25	5	15	6, 7, 8, 4, 6, 5, 2, 3, 6, 7, 8, 6, 7, 5, 7	3,60	einfach
bm34	6	15	6, 6, 6, 5, 4, 7, 5, 3, 3, 4, 8, 8, 7, 3, 3	1,64	einfach
bm42	7	15	6, 7, 8, 7, 8, 5, 4, 3, 6, 5, 3, 4, 7, 6, 5	1,36	einfach
bm52	9	12	4, 5, 6, 5, 4, 7, 8, 6, 7, 5, 6, 4	6,78	einfach
bm60	10	15	4, 3, 5, 7, 6, 5, 6, 4, 3, 5, 8, 7, 6, 5, 4	63,29	mittel
bm62	6	17	3, 3, 3, 3, 4, 4, 4, 4, 4, 5, 5, 4, 4, 4, 3, 4, 3	199,72	schwierig
bm63	6	17	3, 3, 7, 3, 4, 4, 1, 4, 4, 5, 5, 4, 2, 4, 3, 4, 8	86,64	mittel
bm65	7	17	8, 5, 4, 3, 4, 4, 1, 4, 4, 5, 5, 4, 2, 4, 3, 4, 5	966,94	schwierig
bm67	3	16	3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6, 4, 5, 6, 3	1,28	einfach
bm68	4	16	1, 5, 2, 6, 8, 3, 5, 9, 2, 4, 6, 4, 6, 4, 3, 5	8,27	einfach
bm69	5	16	2, 3, 1, 3, 5, 3, 3, 3, 8, 3, 4, 5, 4, 5, 4, 5	56,12	mittel
bm70	6	16	3, 3, 3, 6, 6, 6, 4, 5, 2, 3, 7, 2, 4, 6, 3, 5	43,91	mittel
bm71	7	16	3, 5, 6, 5, 4, 3, 5, 3, 6, 4, 6, 4, 4, 3, 3, 6	1,95	einfach
bm73	9	16	6, 2, 2, 3, 3, 3, 4, 4, 4, 3, 3, 3, 3, 4, 5, 3	110,21	schwierig
bm76	3	17	8, 1, 4, 3, 4, 4, 1, 6, 4, 5, 5, 4, 2, 4, 9, 4, 5	3,67	einfach
bm79	5	17	2, 5, 4, 4, 4, 8, 5, 4, 2, 5, 1, 3, 2, 7, 6, 2, 4	34,02	mittel
bm80	6	17	5, 2, 5, 6, 1, 5, 3, 7, 2, 6, 5, 4, 3, 5, 6, 4, 5	187,74	schwierig
bm81	6	17	5, 4, 5, 3, 2, 5, 3, 4, 2, 3, 5, 4, 6, 5, 6, 2, 4	424,48	schwierig
bm82	7	17	4, 6, 3, 2, 7, 9, 3, 1, 5, 7, 8, 3, 6, 2, 6, 5, 9	29,37	mittel
bm83	7	17	1, 1, 4, 3, 3, 7, 2, 3, 5, 2, 5, 6, 4, 5, 1, 3, 2	123,27	schwierig
bm84	8	17	1, 1, 4, 3, 3, 7, 2, 3, 5, 2, 5, 6, 4, 5, 1, 3, 2	145,34	schwierig
bm85	8	17	6, 3, 4, 7, 2, 6, 8, 9, 2, 4, 1, 5, 7, 4, 6, 4, 2	85,26	mittel
bm86	9	17	4, 4, 2, 3, 3, 3, 4, 6, 4, 5, 5, 7, 6, 6, 6, 3, 3	57,59	mittel
bm87	9	17	9, 4, 2, 5, 3, 8, 4, 6, 4, 5, 5, 7, 6, 2, 6, 3, 3	391,44	schwierig
bm88	10	17	3, 5, 3, 8, 6, 4, 7, 8, 5, 4, 2, 4, 4, 6, 7, 2, 8	99,58	mittel

Tabelle 7.1 Probleme des Benchmarks im Einzelnen. m: *Anzahl Maschinen,* n: *Anzahl Jobs,* t_i: *Bearbeitungszeiten der Jobs,* W: *Anzahl zu expandierender Knoten*[2]

Kennzahlen *(m, n)* bzw. *(b, d)* lassen sich viele triviale Belegungsplanungsprobleme definieren. Zum Beispiel genügt es für alle Jobs, eine identische Bearbeitungszeit zu wählen. Die Lösung dieser Probleme besteht im sukzessiven Einplanen der Jobs auf die Maschinen mit der geringsten Belegungszeit.

Die Kennzahlen *m* und *n* der Probleme legen nur den maximalen Schwierigkeitsgrad fest, der erreicht werden kann. Die entsprechenden Parameter *b* und *d* bestimmen nur die Größe des *unbeschnittenen* Suchbaums. Die Schwierigkeit des Problems ist aber ausschließlich durch die Größe des *beschnittenen* Suchbaums gegeben.

Daher sind die Probleme des Benchmarks so (per Hand) gewählt, daß sie ein Minimum im Schwierigkeitsgrad aufweisen. Damit ein Problem überhaupt ein Problem darstellt müssen mehr Jobs als Maschinen existieren. Es muß also gelten:

[2] Diese Werte stammen aus [Stalp93], Kapitel B.2.5, Verfahren von Mahanti.

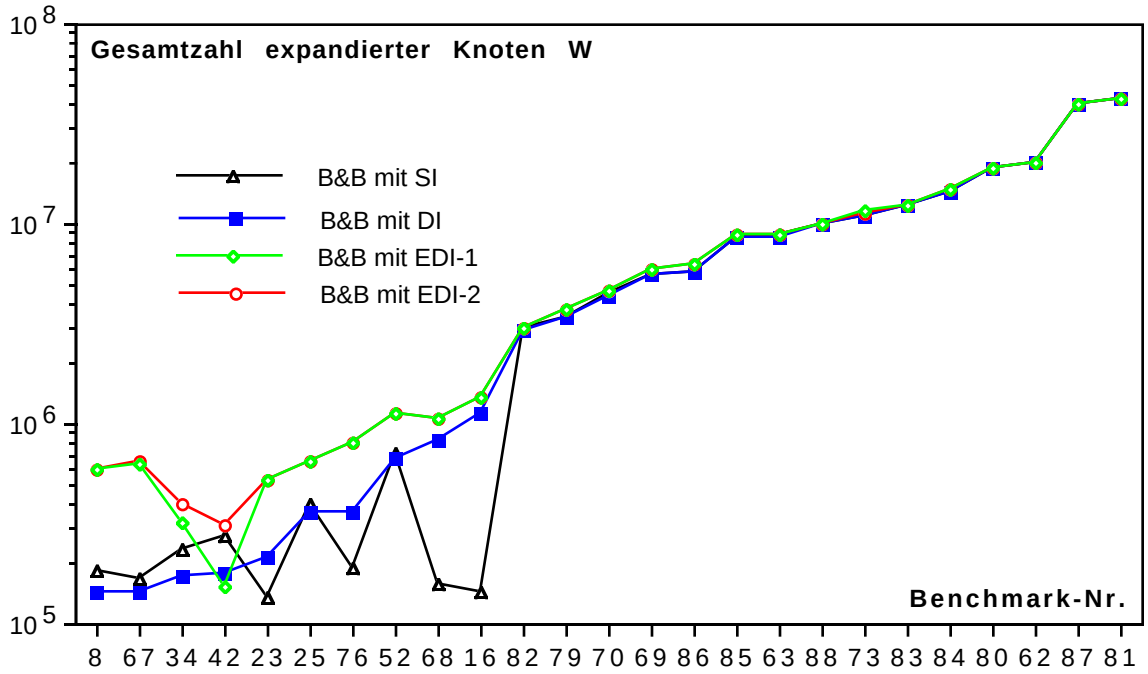

Abb. 7.2: Tatsächlicher Arbeitsaufwand, gemessen in der Anzahl expandierter Knoten W, für die einzelnen Benchmark-Probleme bei unterschiedlichen Initialisierungsmethoden

$$d > b \tag{7.2}$$

Damit sind die in Abb. 7.1 hellgrau unterlegten Probleme von dem Benchmarks ausgeschlossen. Andererseits repräsentieren die dunkelgrau unterlegten Werte Bäume, deren Blattanzahl nicht größer der Prozessoranzahl ist. Das heißt, schon durch die (direkte oder selektive) Initialisierung des B&B mit $P = 2^{14} = 16384$ Prozessoren werden diese Probleme gelöst (Kapitel 3). Damit sind diese Probleme zu einfach und daher uninteressant. Es müssen also nur die Probleme betrachtet werden mit

$$b^d > P \tag{7.3}$$

Bei allen diesen Einschränkungen bleiben für den Schwierigkeitsgrad wichtige Freiheitsgrade offen: Anzahl der Jobs und Maschinen und die Varianz und Verteilung der Bearbeitungszeiten.

Durch die heuristische Beschneidung bei der Baumsuche mit B&B entsteht eine Diskrepanz zwischen der maximalen Problemgröße (unbeschnittener Suchbaum) und der tatsächlich Problemgröße (beschnittener Suchbaum). In Abb. 7.2 sind für unterschiedliche Methoden zur Initialisierung der tatsächliche Arbeitsaufwand aufgetragen. Die Benchmark-Probleme sind hier nach dem Ergebnis der DI aufsteigend sortiert.

Wenn Probleme nach den Bedingungen (7.2) und (7.3) erzeugt werden, dann ist zwar ein Minimum an Aufwand zur Lösung garantiert. Dennoch ist weder bekannt wieviel Aufwand notwendig ist, noch ist eine Sortierung der

Probleme nach ihrer Schwierigkeit möglich. Eine Methode zur Bestimmung des Schwierigkeitsgrads eines Problems ist dessen Lösung. Leider können die parallelen Algorithmen dazu nicht verwendet werden, da gerade sie untersucht werden sollen und ein Bewertungsmerkmal der benötigte Lösungsaufwand, z.B. in Form des tatsächlich abgearbeiteten Suchbaums, darstellt.

Klasse	W	Benchmark-Probleme
zu einfach	$0 - 10^5$	–
einfach	$10^5 - 10^6$	bm8, bm23, bm25, bm34, bm42, bm52, bm67, bm68, bm71, bm76
mittel	$10^6 - 10^7$	bm16, bm60, bm63, bm69, bm70, bm79, bm82, bm85, bm86, bm88
schwierig	$10^7 - 10^8$	bm62, bm65 bm73, bm80, bm81, bm83, bm84, bm87
zu schwierig	$10^8 - \infty$	–

Tabelle 7.2: Klasseneinteilung ausgewählter Benchmark-Probleme nach ihrer Gesamtzahl zu expandierender Knoten W

7.2 Problemabhängige Funktionen

Die in dieser Arbeit verwendeten Anwendungsdomäne ist die non-operationale Belegungsplanung auf parallelen Maschinen (Kapitel 1.5). Die Formulierung des Branch-and-bound-Algorithmus (B&B) aus dem Kapitel 1.2 ist von der Anwendungsdomäne unabhängig, da die einzelnen Probleme mit ihren Unterproblemen zu Knoten abstrahiert sind. Die Gesamtheit der Knoten stellt einen Suchbaum dar, der dann von dem B&B abgearbeitet wird. Da es hier um die problemspezifischen Aspekte des Algorithmus geht, soll nicht mehr von Knoten sondern von Problemen bzw. Teilplänen gesprochen werden.

Für eine bestimmte Anwendung des B&B müssen die problemabhängigen, generischen Funktionen konkretisiert werden. Nach [Lawler66] existieren für das Verfahren elf charakteristische Funktionen. Einige davon dienen zur Zusammenfassung von Teilfunktionen, so daß hier nur vier Funktionen betrachtet werden müssen. Im einzelnen ist das die Expandierungs-, Auswahl-, Bewertungs- und Dominanzfunktion.

Davor wird, zur präziseren Darstellung der in dem B&B-Algorithmus verwendeten, problemabhängigen Funktionen, folgende Notation eingeführt:

m	Anzahl der verfügbaren Maschinen (Resourcen),
n	Anzahl der einzuplanenden Aufträge (Jobs),
$\underline{I}$	$= \{1, \ldots, m\}$, Indexmenge aller einplanbaren Maschinen,
$\underline{J}$	$= \{1, \ldots, n\}$, Indexmenge aller einzuplanenden Aufträge,
t_j	Bearbeitungszeit des Auftrags $j \in \underline{J}$,
x, y, z	(unvollständige oder partielle) Pläne (schedules),
$J(x)$	Indexmenge der im Plan x eingeplanten Aufträge $(J(x) \in \underline{J})$,
$J^c(x)$	Indexmenge der nicht in x eingeplanten Aufträge $(J^c(x) \cup J(x) = \underline{J})$,
$V(x)$	Indexmenge der in x einplanbaren (verfügbaren) Jobs $(V(x) \in J^c(x))$
$M(x, i)$	Belegungszeit der Maschine i im Teilplan x,
$M(x)$	$= \max\{M(x, i)\}$ für $i \in \underline{I}$, Gesamtbearbeitungszeit (makespan)

7.2.1 Expansionsfunktion

Die Expansionsfunktion (expansation function) gibt an, wie ein Problem vollständig in disjunkte Teilprobleme zerlegt wird. Das heißt, die erzeugten Teilprobleme repräsentieren komplett das Ursprungsproblem. Das Problem kann dann durch seine Teilprobleme ersetzt und muß selbst nicht mehr betrachtet werden. Die Teilprobleme stellen entweder Lösungen des Ursprungsproblems dar, oder sie werden ihrerseits wieder expandiert. In dem Algorithmus aus Kapitel 1.2 steckt die Expansionsfunktion in der Bedingung der for-Schleife: "`all successors y of x`".

Bei Problemen der Belegungsplanung plant die Expansionsfunktion in einen gegebenen Teilplan einen weiteren Auftrag ein. Falls nicht mehrmals der gleiche Plan erzeugt wird, sind die erzeugten Pläne disjunkt. Aber es gilt noch festzulegen, welcher Auftrag auf welche Maschine eingeplant wird. Bei dieser Festlegung muß beachtet werden, daß durch die Expansion potentiell der vollständige Suchbaum aufgespannt wird. Falls also alle erzeugten Unterprobleme weiterverfolgt würden, dann könnten alle möglichen (vollständigen) Pläne aufgezählt werden.

Für die obige Problemstellung mit parallelen Maschinen kann man mehrere verschiedene Expansionsfunktionen angeben. Die erste Regel $e_0(x)$ plant in einen gegebenen Teilplan x alle verfügbaren Aufträge auf alle Maschinen der Reihe nach ein. Bei n Aufträgen und m Maschinen werden bei jeder Expansion $n \cdot m$ neue Teilpläne erzeugt:

$$e_0(x) := \{y_{1,1}, \ldots, y_{m,n} \mid y_{i,j} = \text{"Plan } x \text{ mit Job } j \text{ auf Maschine } i \text{ eingeplant"}\} \quad (7.4)$$

Die Funktion $e_0()$ ergibt einen sehr breiten Baum. Nach der Problemdefinition ist die Reihenfolge der Aufträge auf den einzelnen Maschinen nur durch die Vorrangbedingungen eingeschränkt. Für das hier gewählte Optimierungskriterium "Gesamtbearbeitungszeit" ist die Reihenfolge unerheblich. Daher kann die Breite des Suchbaums eingeschränkt werden. Eine weitere Gruppe von Expansionsfunktionen plant genau einen Auftrag auf alle Maschinen ein. Es werden also immer m Teilpläne erzeugt.

$$e(x) := \{y_1, \ldots, y_m \mid y_i = \text{"Plan } x \text{ mit Job } k \text{ auf Maschine } i \text{ eingeplant"}, \; t_k = \mathrm{OP}_{j \in V(x)}\{t_j\}\} \quad (7.5)$$

Die Expansionsfunktion kann noch weiter verbessert werden. Bei der Optimierung wird die Gesamtbearbeitungszeit mit identischen Maschinen minimiert. Daraus ergibt sich, daß mehrere Maschinen mit gleicher (partieller) Belegungszeit für die Optimierung nicht unterschieden werden müssen. Die Expansionsfunktion muß also einen Job nur auf jeweils einer Maschine mit unterschiedlicher Bearbeitungszeit einplanen. Der Verzweigungsfaktor des Suchbaums hängt damit von der Problemstellung ab, ist aber immer kleiner m:

$$e(x) := \{y_{l_1}, \ldots, y_{l_m} \mid y_{l_i} = \text{"Plan } x \text{ mit Job } k \text{ auf Maschine } l_i \text{ eingeplant"}, \; t_k = \mathrm{OP}_{j \in V(x)}\{t_j\}, \; l_i \in \{i \in \{1,\ldots,m\} \mid \mathrm{M}(x, i) \neq \mathrm{M}(x, i'), \; i \neq i'\}\} \quad (7.6)$$

Festzulegen bleibt, welcher Auftrag zur Einplanung ausgewählt wird. Dies wird in der Funktion OP{} als Platzhalter für drei konkrete Operationen getan. In der Expansionsfunktion $e_1(x)$ wird der verfügbare Auftrag mit der längsten Bearbeitungszeit eingeplant. Die Funktion $e_2(x)$ dagegen bevorzugt den Auftrag mit der kürzesten Bearbeitungszeit. Auch hier muß dieser Auftrag verfügbar sein, d.h. alle seine Vorgängeraufträge sind schon eingeplant. Schließlich wird in einer letzten Expansionsfunktion $e_3(x)$ ein verfügbarer Auftrag rein zufällig ausgewählt. Zusammenfassend ergibt sich folgende Zuordnung:

$$e_1(x)\text{: } e(x) \text{ mit OP}\{\} = \max\{\}, \tag{7.7}$$

$$e_2(x)\text{: } e(x) \text{ mit OP}\{\} = \min\{\}, \tag{7.8}$$

$$e_3(x)\text{: } e(x) \text{ mit OP}\{\} = \text{random}\{\}. \tag{7.9}$$

Schon bei der heuristisch berechneten, ersten Lösung ergibt eine Auswahl der Jobs nach $e_1(x)$ sehr gute Pläne. Daher wurde diese Expansionsfunktion auch in den Experimenten mit dem B&B-Algorithmus verwendet.

7.2.2 Bewertungsfunktion

Zu dem Minimierungskriterium $f(x)$ aus der Problemstellung gibt die Bewertungsfunktion $g(x)$ (bounding function) eine untere Schranke des Kriteriums an. Das heißt, für einen bestimmten Teilplan x ist die Bewertungsfunktion eine Abschätzung nach unten für die entstehenden Kosten: $g(x) \leq f(x)$. Ist ein Plan x eine gültige Lösung, d.h. alle Jobs sind eingeplant, dann hat die Bewertungsfunktion den Wert des Minimierungskriteriums $(g(x) = f(x))$. Im B&B-Algorithmus wird diese Funktion eingesetzt, um durch Beschneidung des Suchbaums effizient eine Lösung zu finden. Bei einer perfekten Bewertung aller Teilpläne, d.h. $g(x) = f(x)$, würde der B&B im Suchbaum auf direktem Weg zu einem Optimum finden.

Für die Bewertungsfunktion sind hier mehrere Varianten denkbar. Eine triviale untere Schranke $g_0(x)$ eines Teilplans x ist seine bisherige Gesamtbearbeitungszeit. Im Suchbaum des B&B entspricht dies dem zurückgelegten Weg zum dazugehörigen Teilplan ohne Abschätzung des Restweges bis zum vollständigen Plan:

$$g_0(x) := M(x) \tag{7.10}$$

Eine zweite, leicht einzusehende Bewertungsfunktion verteilt die verbleibende Arbeit ideal auf die vorhandenen Maschinen. Dabei werden mögliche Vorrangbedingungen nicht berücksichtigt und u.U. die Aufträge entgegen der Problemdefinition unterbrochen. Zur Berechnung dieser unteren Schranke werden zwei Summen benötigt. Zum einen ist dies die verbleibende Arbeit u mit

$$u = \sum_{j \in J^c(x)} t_j$$

Zum anderen ist die Summe der ungenützten Maschinenzeiten v. Hierbei wird diese seit dem letzten Job auf einer Maschinen bis zur längsten Bearbeitungszeit einer Maschine gerechnet:

$$v = \sum_{i \in \underline{I}} M(x)\text{-}M(x, i)$$

Die zweite Bewertungsfunktion ergibt sich dann aus der ersten unteren Schranke mit der berechneten idealen Verteilung:

$$g_1(x) := g_0 + \left\lfloor \frac{\max(u - v, 0)}{m} \right\rfloor \tag{7.11}$$

Zusätzlich kann man extrem lange Jobs mit berücksichtigen. Sie sind nach der Problemdefinition nicht unterbrechbar. Dazu wird die Bearbeitungszeit s des längsten, noch nicht eingeplanten Jobs und die kürzeste der Maschinenarbeitszeiten r benötigt. Sie ergeben sich aus: $s = \max\{t_j\}$ für $j \in J^c(x)$ und $r = \min\{M(x, i)\}$ für $i \in I$. Damit kann die neue Bewertungsfunktion als Erweiterung der vorigen unteren Schranke angegeben werden:

$$g_2(x) := \max\{g_1(x), r + s\} \tag{7.12}$$

Hier wurde in dem B&B-Algorithmus die Bewertungsfunktion g_2 verwendet. Eine weitere, denkbare Funktion geht auf die möglichen Reihenfolgebedingungen der Aufträge ein. Für die Reihenfolge der Aufträge ist der kritische Pfad ein wichtiges Merkmal. Er besteht aus denjenigen Aufträgen, deren Verzögerung eine Verlängerung der gesamten Bearbeitungszeit nach sich ziehen würde. Er gibt damit die kleinste Zeit an, die mindestens notwendig ist, um alle Aufträge zu bearbeiten. Zur Berechnung des kritischen Pfads siehe [Müller70]. In der Bewertungsfunktion wird die Länge des kritischen Pfades zu der aktuellen Bearbeitungszeit addiert. Diese untere Schranke schließt $g_2(x)$ mit ein, da der längste Auftrag ohne Vorrangbeziehungen auch einen kritischen Pfad bildet.

7.2.3 Auswahlfunktion

Bei dem B&B-Algorithmus werden in jeder Iteration die expandierten und brauchbaren Teilpläne in die Menge OPEN eingefügt. In den nachfolgenden Iterationen muß wiederum ein zu expandierenden Teilplan x aus dieser Menge ausgewählt werden. Dies geschieht durch Minimieren einer Auswahlfunktion (selection function). Der Teilplan mit dem kleinsten Wert der Funktion wird ausgewählt: $h(x) = \min$. In den meisten Fällen müssen mehrere Auswahlkriterien in diese Funktion eingebaut werden, um eine eindeutige Planauswahl zu garantieren. Die Kriterien haben unterschiedliche Priorität und werden der Reihe nach angewendet:

$$h(x) = (h'(x), h''(x), \ldots) \tag{7.13}$$

Tritt bei dem ersten Kriterium eine Mehrdeutigkeit auf, so wird das Kriterium mit der nächst niederen Priorität zur Auswahl eingesetzt. Dies geschieht solange, bis genau ein Teilplan zur Weiterbearbeitung identifiziert wurde. In dem Algorithmus aus Kapitel 1.2 steckt die Auswahlfunktion in der Anweisung "`x = select(OPEN)`".

Üblicherweise wird als erstes Kriterium entweder der Teilplan mit der kleinsten Bewertung (Bestensuche) oder mit der größten Tiefe im Suchbaum

(Tiefensuche) ausgewählt. Bei der Bestensuche ist $h'(x)$ die untere Schranke für die Kosten des Teilplans und wird durch die Bewertungsfunktion berechnet. Mehrdeutigkeiten wurden durch Anzahl eingeplanter Jobs $|J(x)|$ (Tiefe im Suchbaum) des Teilplans x aufgelöst. Damit ist Auswahlfunktion $h_{BF}(x)$ für die Bestensuche gegeben durch:

$$h_{BF}(x) := (g(x), -|J(x)|) \tag{7.14}$$

Dagegen wird bei der Tiefensuche $h'(x)$ erst durch die negative Tiefe $-|J(x)|$ des Teilplans x im Suchbaum bestimmt. Bei Mehrdeutigkeiten kommt die Iteration $i(x)$, in welcher der Teilplan x erzeugt wurde zum Tragen. Damit richtet sich die Auswahl nach der natürlichen Numerierung für Suchbäume[3]. Die Auswahlfunktion $h_{DF}(x)$ für die Tiefensuche ist dann:

$$h_{DF}(x) := (-|J(x)|, i(x)) \tag{7.15}$$

7.2.4 Dominanzrelation

Diese Funktion (dominance relation) dient dazu einen Teilplan x gegenüber einem anderen Teilplan y als dominant zu kennzeichnen. Dabei bedeutet "dominant", daß y bei Betrachtung von x nicht mehr berücksichtigt werden muß und verworfen werden kann. Dieser Fall tritt z.B. auf, wenn x alle Lösungen beinhaltet, zu welchen auch der Teilplan y führen kann. Da die Dominanzrelationen sehr anwendungsabhängig sind, wurden sie in dieser Arbeit nicht angewendet.

---

[3] Natürliche Numerierung von Suchbäumen: Von der Wurzel beginnend, ebenenweise, von links nach rechts.

7.3 Laufzeiten der Grundoperationen

Hier werden Laufzeitmessungen von Grundoperationen des parallelen Branch-and-bound (B&B) beschrieben. Sie dienen zum relativen Aufwandsvergleich der Operationen in den verschiedenen Lastverteilungsvarianten. Eine Darstellung des B&B-Algorithmus bzw. die Funktionsweise von SIMD-Rechnern wird in Kapitel 1.2 und 1.3 gegeben. Für eine Beschreibung des für die Experimente verwendeten Rechners MasPar MP-1 siehe [MasPar92a]. Als Anwendungsbeispiel wird das Problem der Maschinenbelegungsplanung betrachtet (Kapitel 1.5).

In der hier zugrunde gelegten Implementierung beträgt der Speicherbedarf eines Suchbaumknotens 236 Bytes. Bei einer Speicherkapazität von 16 KByte Speicher pro Prozessor können unter Berücksichtigung des Speicherbedarfs des Laufzeitsystems maximal 50 Knoten lokal gespeichert werden. Außerdem handelt es sich bei der verwendeten Implementierung um einen Prototypen, der eine starke Flexibilität in den Varianten aufweist. Das heißt, die hier ermittelten Zeitkonstanten der Grundoperationen können für eine gegebene Anwendung durch weitere Optimierung u.U. verringert werden.

Für die Zeitmessung sind in der Programmbibliothek (MPL-Library) mehrere Funktionen vorgesehen. Ein Zyklus der internen Uhr heißt *Tick* und benötigt 80 ns, d.h. eine Millisekunde entspricht 12.500 Ticks. Da alle zu messende Operationen in einer Schleife mit 1.000 Wiederholungen gemessen wurden, steigert sich die theoretische Genauigkeit von ± 80 ns auf ± 0,08 ns. Aber bei wiederholtem Ablauf einer Messung zeigt sich, daß die durch das Laufzeitsystem entstehende Ungenauigkeit weit größer ist als dieser theoretische Wert. Die geschätzte Genauigkeit der Zeitmessung liegt bei etwa 100 ns.

Im Rest dieses Unterkapitels werden einerseits globale und lokale Kommunikationsmechanismen behandelt und andererseits die grundlegenden Operationen auf der Datenstruktur OPEN untersucht. Zu den Grundoperationen des B&B gehören weiterhin neben der Expansion eines Knotens und Generierung seiner Nachfolger auch die heuristische Bewertung von Knoten in Form von unteren Schranken der Kosten. Im folgenden wird für jede dieser Grundoperationen der Laufzeitaufwand experimentell bestimmt und in Form von Konstanten bzw. Funktionen verallgemeinernd zusammengefaßt.

7.3.1 Kommunikation von Knoten

Lokale Kommunikation

In der MasPar MP-1 kann über das sogenannte X-Net mit Hife der Funktion `pp_xsend()` in acht Richtungen (Nord, Nordost, ..., Nordwest) gleichwertig kommuniziert werden. Die Prozessoren haben also eine 8-Nachbarschaft. Bei der

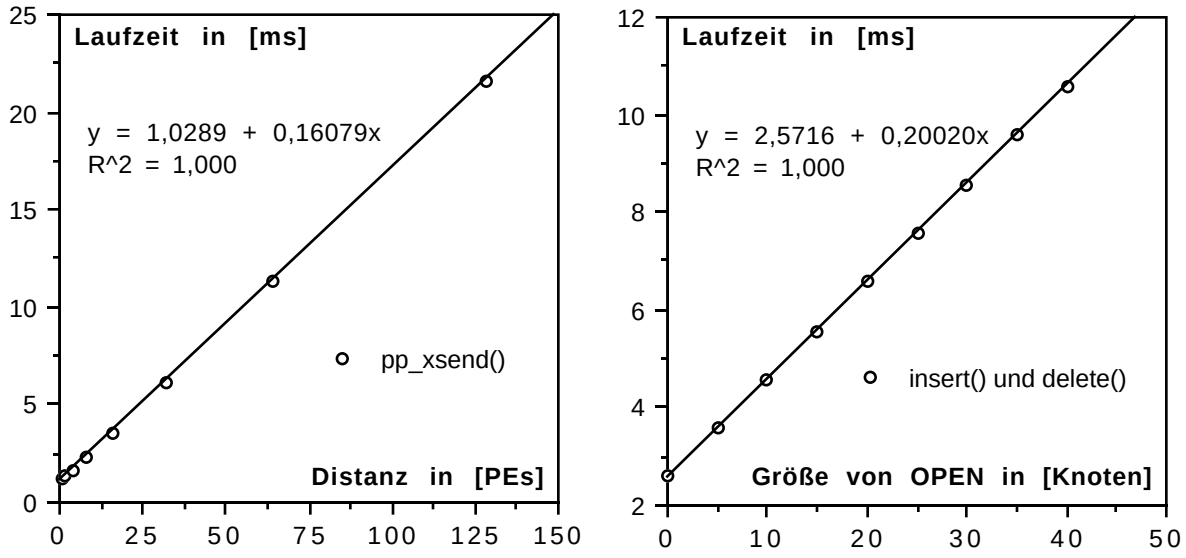

Abb. 7.4: Laufzeiten von `pp_xsend()` (links) und von `insert()` bzw. `delete()` (rechts)

Messung von `pp_xsend()` wird ein Knoten des Suchbaums von einem Prozessor in die Richtung *(dx, dy)* zu einem anderen Prozessor gesendet. Dabei wird die Distanz *dx* in der Anzahl überbrückter Prozessoren gemessen. Hier wird der Einfachheit wegen für die Messungen $dx = dy$ gewählt. Da die zu Verfügung stehende MP-1 ein Prozessorfeld der Größe 128x128 Prozessoren hat, ist die maximale Distanz von 127 Prozessoren ausreichend.

In Abb. 7.4 sind links die Zeitmessungen für exponentiell ansteigende Distanzen angegeben. An die Meßwerte ist eine Gerade angepaßt. Es ist zu erkennen, daß sich das Zeitverhalten von `pp_xsend()` durch eine lineare Funktion sehr genau modellieren läßt. In der Abbildung ist zusätzlich die Geradengleichung für die Laufzeit der Transferoperation abhängig von der Distanz $x \geq 1$ der Prozessoren angegeben.

Globale Kommunikation

Im Gegensatz zur lokalen Kommunikation aus dem letzten Abschnitt sind die hier besprochenen Mechanismen global. Die globale Kommunikation kann zum einen über den sogenannten Router bewerkstelligt werden. Dabei können mehrere Prozessoren (mehr oder weniger parallel) Daten zu einem Empfänger senden oder von einem Sender Daten empfangen. Andererseits kann auch der Datenbus eingesetzt werden, der alle Prozessoren und die Array Control Unit (ACU) verbindet.

Im B&B wird die globale Kommunikation z.B. für das Versenden von Knoten an alle Prozessoren (Broadcast) benötigt. Dafür wird bei der MP-1 ein Knoten von einem bestimmten Prozessor auf die ACU über den Datenbus transferiert. Dann kann von der ACU an alle Prozessoren gleichzeitig über den

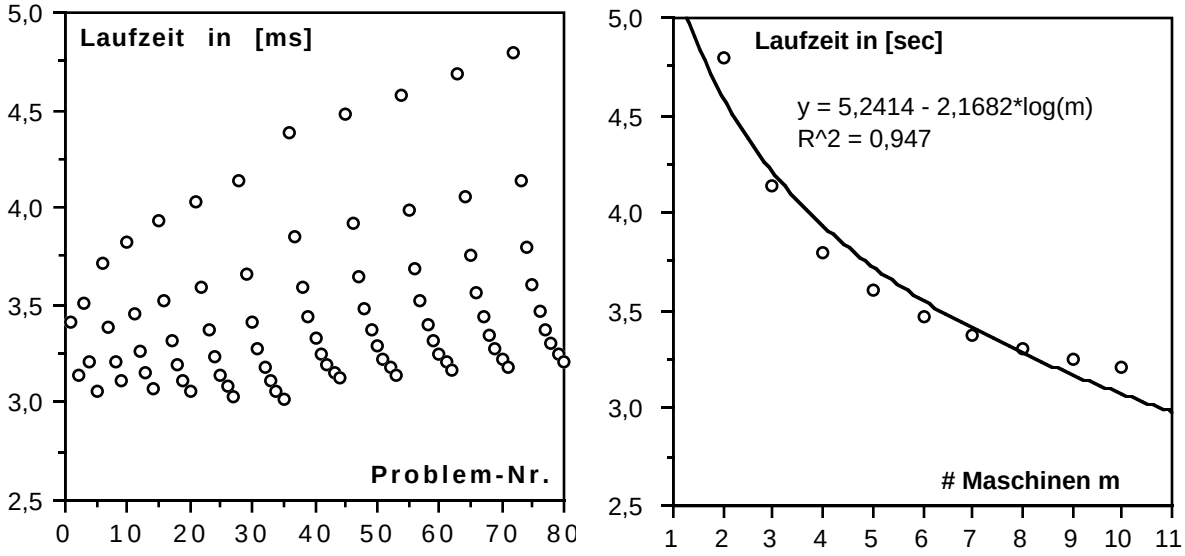

Abb. 7.5: Laufzeiten für die Expandierungsfunktion für unterschiedliche Probleminstanzen (links) und ein Ausschnitt davon mit analytischer Approximation (rechts)

Datenbus der Knoten versendet werden. Die für den Broadcast benötigte Zeit beträgt 4,85 ms und ist unabhängig von dem Index des sendenden Prozessors.

Weitere globale Kommunikation sind im B&B denkbar. Zum Beispiel für die Lastverteilung mit der Rendezvouz-allocation-method muß der Router eingesetzt werden [Mahanty93, Powley93, Karypis92]. Die Laufzeiten der Router-Kommunikation mit verschiedenen Kommunikationsmustern wurde für die MasPar MP-1 in [Prechelt93] untersucht.

7.3.2 Operationen auf OPEN

Weitere grundlegende Operationen sind die Zugriffe auf die Datenstruktur OPEN. Da der lokale Speicher pro Prozessor relativ klein ist, wurde OPEN als sortierte Liste mit bis zu 50 Elementen implementiert. Dies ist zwar asymptotisch betrachtet nicht die effizienteste Datenstruktur, aber dafür ist der Verwaltungsaufwand sehr klein. Die Liste ist in einem Feld mit Zeigern auf die Knoten links beginnend angelegt. Der Knoten mit der schlechtesten Bewertung befindet sich am linken Ende der Liste. Somit müssen bei den häufigsten Operationen im B&B (`insert`() und `delete`()) nur wenige Knoten in dem Feld verschoben werden.

Bei jeder B&B-Iteration wird ein Knoten aus OPEN entfernt und ein anderer eingefügt (`insert`() und `delete`()). Für die Messung der Laufzeit werden die zwei Operationen zu einem Zugriff zusammengefaßt und in dieser Form gemessen (Abb. 7.4, rechts). Die Länge der Liste steigt jeweils um fünf Knoten. Die Werte der Knoten sind so gewählt, daß der einzufügende Knoten immer an das linke Ende der Liste angehängt wird. Es müssen also alle Knoten verschobenen werden. Dieser für eine gegebene Größe von OPEN ungünstigste Fall tritt bei

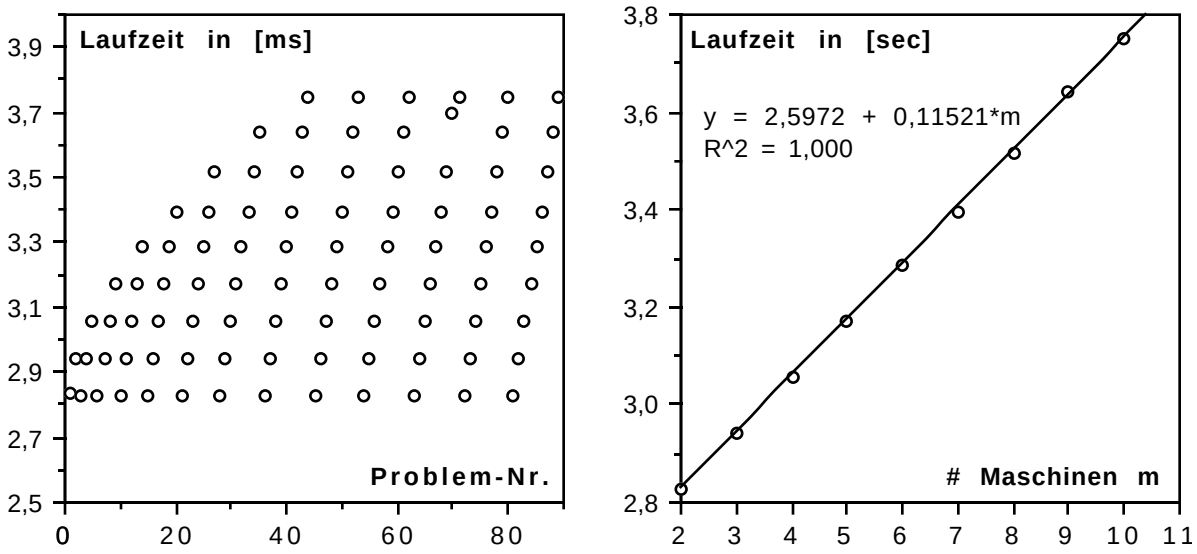

Abb. 7.6: Laufzeiten der Bewertungsfunktion $g()$ für unterschiedliche Probleminstanzen (links) und ein Ausschnitt davon mit analytischer Approximation (rechts)

SIMD-Rechnern mit großer Wahrscheinlichkeit ein, denn mit vielen Prozessoren existiert sicherlich ein lokales OPEN, bei dem alle Knoten verschoben werden müssen. Die restlichen Prozessoren müssen auf Grund des Look-step-Modus der SIMD-Rechner auf diesen einen Prozessor warten.

7.3.3 Expansion von Knoten

Innere Knoten des Suchbaums, die noch keine vollständige Lösung darstellen, werden expandiert. Innerhalb einer B&B-Iteration werden alle Nachfolger des aktuellen Knotens mit Hilfe der Expansionsfunktion erzeugt. Auch diese Grundoperation ist wie die Bewertung der Knoten anwendungsabhängig und wird daher nur für unser Belegungsplanungsproblem betrachtet. In Abb. 7.5 sind links für verschiedene Problemstellungen die Laufzeiten aufgetragen, die benötigt werden, um alle Nachfolger der Wurzel zu erzeugen. Der Zeitaufwand ist wiederum von dem Problem abhängig. Die Probleminstanzen sind sortiert nach (1) aufsteigender Anzahl von Aufträgen ($n = 3, \ldots, 14$) und (2) aufsteigender Anzahl von Maschinen ($m = 2, \ldots, \min\{n, 10\}$).

Die Meßpunkte bilden zwölf Gruppen, welche jeweils auf einem nichtlinearen abfallenden Funktionsgraph liegen. Die Meßwerte jeder Gruppe stammen von Problemen mit derselben Anzahl von Jobs n. Innerhalb einer Gruppe stammen die ansteigenden Meßwerte von Problemen mit zunehmender Maschinenanzahl m. An die letzte Gruppe (Problem-Nr. 72 bis 80) kann als erste Annäherung eine logarithmische Funktion abhängig von m angepaßt werden (Abb. 7.5, rechts).

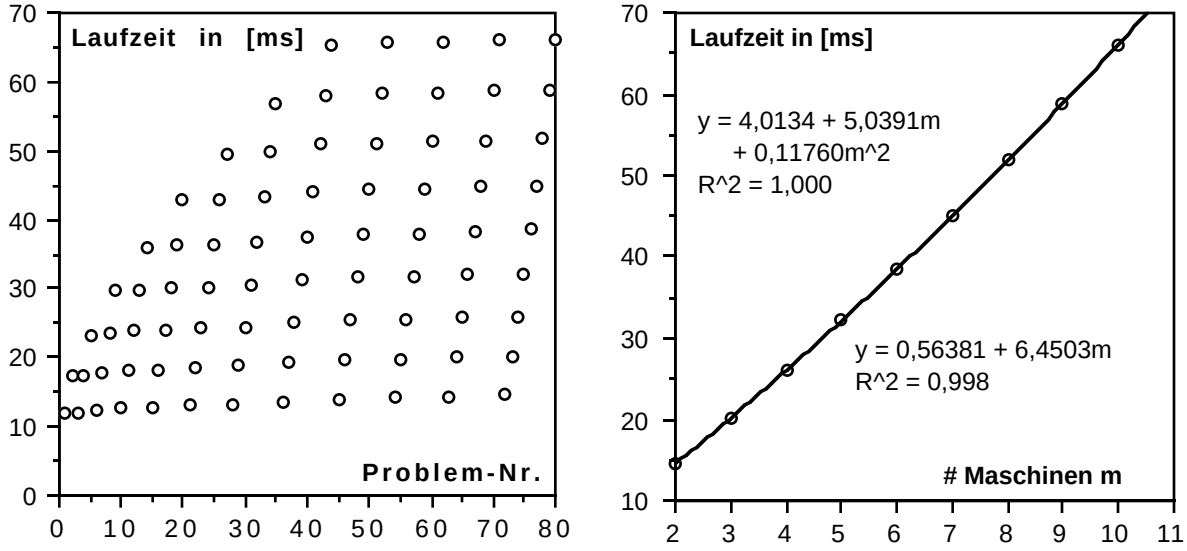

Abb. 7.7 Laufzeiten einer Branch-and-bound-Iteration ohne OPEN-Operationen für unterschiedliche Probleminstanzen (links) und ein Ausschnitt davon mit analytischer Approximation (rechts)

7.3.4 Bewertung von Knoten

In dem B&B wird für jeden inneren Knoten (Teillösung) eine untere Schranke für die Kosten der daraus entstehenden vollständigen Lösung mit Hilfe der Bewertungsfunktion *g()* berechnet. Diese Berechnung ist stark anwendungsabhängig und wird daher nur für unser Belegungsplanungsproblem betrachtet. Die Berechnungszeiten variieren je nach Problemgröße etwas. In Abb. 7.6 links bilden unterschiedlich große Probleminstanzen die *x*-Achse. Darüber aufgetragen sind die Laufzeiten zur Evaluierung der Wurzel des jeweiligen Problems. Auch hier sind die Probleminstanzen sortiert nach (1) aufsteigender Anzahl von Aufträgen ($n = 3, \ldots,15$) und (2) aufsteigender Anzahl von Maschinen ($m = 2, \ldots, \min\{n, 10\}$). Die Meßwerte zeigen eine starke Abhängigkeit von der Maschinenanzahl *m* und kaum eine Abhängigkeit von der Jobanzahl *n* des Problems.

Wiederum bilden die Meßpunkte zwölf Gruppen, die diesmal annähernd auf jeweils einem linearen Funktionsgraph liegen. Die Meßwerte jeder Gruppe stammen von Problemen mit derselben Anzahl von Jobs *n*. Innerhalb einer Gruppe stammen die ansteigenden Meßwerte von Problemen mit zunehmender Maschinenanzahl *m*. An die letzte Gruppe kann eine Gerade abhängig von der Maschinenanzahl *m* angepaßt werden (Abb. 7.6, rechts).

7.3.5 Eine B&B-Iteration

Die Kombination von einigen Knotengenerierungen, Evaluationen und OPEN-Operationen ergibt eine Iteration des B&B. Nun werden von der

Problemgröße abhängenden Laufzeiten betrachtet. Davon nicht betroffen sind die OPEN-Operationen, da sie nur von der Länge der Liste abhängig sind. In Abb. 7.7 sind links, analog zu Abb. 7.5 und Abb. 7.6, die Probleme entlang der *x*-Achse aufgelistet. Darüber aufgetragen sind die Laufzeiten für die Generierung und Evaluation aller Nachfolger der Wurzel des jeweiligen Suchbaumes.

Auch hier bilden die Meßpunkte zwölf Gruppen mit Meßpunkten auf nicht-linearen Funktionsgraphen. An die letzte Gruppe kann die quadratische Funktion abhängig von der Maschinenanzahl *m* angepaßt werden (Abb. 7.7 rechts). Insgesamt kommt sicherlich noch ein kleiner Summand für die wachsende Anzahl von Jobs hinzu. Vereinfachend soll die Anpassung einer linearen Funktion verwendet werden.

7.3.6 Zusammenfassung

Die oben beschriebenen Messungen von Grundoperationen des B&B wurden unter folgenden Voraussetzungen durchgeführt:

- Rechnerarchitektur: SIMD-Rechner mit indirekter Adressierung, MasPar MP-1
- Anwendungsdomäne: Belegungsplanung mit parallelen, identischen Maschinen ohne Vorrangbedingungen
- Implementierung: OPEN als sortierte Liste mit Zeiger auf die Elemente.
- Meßgenauigkeit: 100 ns (geschätzt)

Zusammenfassend ergeben die Messungen Ergebnisse in Tabelle 7.3.

Art der Grundoperation:	Bez.:	Laufzeit in [ms]:
Transfer eines Suchbaumknotens übers X-Net zwischen zwei in einer Linie liegende Prozessoren mit Abstand *x* :	T_{trans}	$1{,}03 + 0{,}16 \cdot x$
Versenden eines Knotens von einem Prozessor an alle Prozessoren (Broadcast):	T_{send}	4,85
Einsortieren und Entnehmen eines Knotens in die Datenstruktur OPEN mit *x* enthaltenden Elementen:	T_{open}	$2{,}57 + 0{,}20 \cdot x$
Generierung eines Nachfolgerknotens bei einem Verzweigungsgrad von $m \geq 2$ Maschinen:	T_{gen}	$(5{,}24 - 2{,}17 \cdot \log(m)) / \text{m}$
Evaluierung eines Knotens bei einem Verzweigungsgrad von $m \geq 2$ Maschinen:	T_{eval}	$2{,}60 + 0{,}12 \cdot m$
Eine standardmäßige Iteration ohne OPEN-Operationen bei einem Verzweigungsgrad von $m \geq 2$ Maschinen:	T_{iter}	$0{,}56 + 6{,}45 \cdot m$
Ein einzelner Reduktionsschritt zur Minimumsbildung über 4 Byte bei $P = 16.384$ Prozessoren (hochgerechnet):	T_{red}	$0{,}16 / P$

Tabelle 7.3: Zusammenstellung der analytisch approximiertern Laufzeiten für die Grundoperationen des parallelen Branch-and-bound auf der MasPar MP-1

8. Register

8.1 Literaturverzeichnis

[Abdelrahman88] Abdelrahman T. S., Mudge T. N., 1988, "Parallel branch and bound algorithms on hypercube multiprocessors", Proc. of the 1988 ACM Conference on Lisp and Functional Programming, pp 1492-1499.

[Akl89] Akl S. G., 1989, "The design and analysis of parallel algorithms", Prentice-Hall.

[Arvindam89] Arvindam S., Kumar V., Rao V. N., 1989, "Floorplan optimization on multiprocessors", Proc. of th 1989 Int. Conf. on Computer Design.

[Augustin93] Augustin B., 1993, "Eine Simulationsumgebung für Scheduling-Algorithmen", Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe.

[Baker74a] Baker K. R., 1974, "Introduction to sequencing and scheduling", Wiley, New York.

[Baker74b] Baker K. R., Su Z.-S., 1974, "Sequencing with due-dates and early start times to minimize maximum tardiness", Naval Res. Logist. Quart., vol 21, pp 171-176.

[Barnes77] Barnes J. W., Brennan J. J., 1977, "An improved algorithm for scheduling jobs on identical machines", AIIE Trans., vol 9, no 1, pp 25-31.

[Bertsekas89] Bertsekas D. P., Tsitsiklis J. N., 1989, "Parallel and distributed computation: Numerical methods", Prentice-Hall, Englewood Cliffs, NJ, pp 519-526.

[Biagioni91] Biagioni E. S., Prins J., 1991, "Scan-directed load balancing for highly-parallel mesh-connected computers", Bibliographie der MasPar Computer GmbH, MP/PA-12.92.

[Boillat90] Boillat J. B., 1990, "Load balancing and poisson equation in a graph", Concurrency: Practice and Experience, vol 2, no 4, pp 289-313.

[Casavant88] Casavant T. L., Kuhl J. G., 1988, "A taxonomy of scheduling in general-purpose distributed computing systems", IEEE Trans. on Software Engineering, vol 14, no 2, pp 141-154.

[Cheng90] Cheng T. C. E., Sin C. C. S., 1990, "A state-of-the-art review of parallel-machine scheduling research", European Journal of Operational Research, vol 47, pp 271-292.

[Christman83] Christman D. P., 1983, "Programming the Connection Machine", Master's Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

[Cybenko89] Cybenko G., 1989, "Load balancing for distributed memory multiprocessors", Jour. Parallel Distributed Comput., vol 7, pp 279-301.

[Dehne90] Dehne F., Ferreira A. G., Rau-Chaplin A., 1990, "Parallel AI algorithms for fine-granind hypercube multiprocessors", Proc. V. Int. Ws. on Parallel Processing by Cellular Automata and Arrays, Berlin, pp 51-65.

[Duden88] Duden-Informatik, 1988, B.I.-Wissenschaftsverlag.

[ElDessouki80] El-Dessouki O., Huen W. H., 1980, "Distributed enumeration on network computers", IEEE Trans. on Computers, vol 29, pp 818-825.

[Elmaghraby74] Elmaghraby S. E., Park S. H., 1974, "Scheduling jobs on a number of identical machines", AIIE Trans., vol 6, no 1, pp 1-13.

[Evett90] Evett M., Hendler J., Mahanti A., Nau D., 1990, "PRA*: A memory-limited heuristic search procedure for the Connection Machine", The 3rd Symposium on the Frontiers of Massively Parallell Computation, Okt., College Park, Maryland.

[Felten88] Felten E. W., 1988, "Best-first branch-and-bound on a hypercube", Proc. of the 3rd Conf. on Hypercube, Concurrent Computers, and Applications, Pasadena, CA.

[Flynn72] Flynn M. J., 1972, "Some computer organizations and their effectiveness", IEEE Trans. on Computers, vol C-21, no 9, pp 948-960.

[Frye90] Frye R., Myczkowski J., 1990, "Exhaustive search of unstructured trees on the Connection machine", Thinking Machines Corporation Technical Report TMC-196, Cambridge, MA.

[Garey79] Garey M. R., Johnson D. S, 1979, "Computers and intractability - A guide to the theory of NP-completeness", W. H. Freeman and Company, New York.

[Gerogiannis93] Gerogiannis D., Orphanoudakis S. C., 1993, "Load balancing requirements in parallel implementations of image feature extraction tasks", IEEE Trans. of Parallel and Distributed Systems, vol 4, no 9, pp 994-1013.

[Gonauser89] Gonauser M., 1989, "Multiprozessor-Systeme", Springer-Verlag.

[Gustafson88] Gustafson J. L., 1988, "Reevaluation Amdahl's law", Communications of the ACM, vol 31, no 5, pp 532-533.

[Heiss93] Heiss H.-U., Schmitz M., 1993, "Decentralized dynamic load balancing: The particles approach", 8th Int. Symp. on Computers and Information Science (ISCIS VIII), Istanbul.

[Hemminger94] Hemminger J., 1994, "Eine lokale Lastverteilung für daten-parallelen Branch-and-bound", Studienarbeit, Universität Karlsruhe.

[Henrich93] Henrich D., 1993, "Initialization of parallel branch-and-bound algorithms", 2nd Int. Workshop on Parallel Processing for Artificial Intelligence (PPAI-93), August 29, Chambery, France (to be published by Elsevier in 1994).

[Henrich94a] Henrich D., 1994, "Lokale Lastverteilung für daten-paralleles Branch-and-bound", 22. Workshop Komplexität und effiziente Algorithmen, 8. Februar, Saarbrücken, Technischer Bericht des Max-Planck-Institut für Informatik, Ed: R. Fleischer, MPI-I-94-104.

[Henrich94b] Henrich D., 1994, "Local load balancing for data parallel branch-and-bound", International Conference Massively Parallel Processing, June 21-23, Delft, The Netherlands.

[Henrich94c] Henrich D., 1994, "The Liquid Model Load Balancing Method", Journal of Parallel Algorithms and Applications, Special Issue on Algorithms for Enhanced Mesh Architektures. - *eingereicht*.

[Henrich94d] Henrich D., 1994, "Paralleles Branch-and-bound und Scheduling", Universität Karlsruhe, Fakultät für Informatik, Interner Bericht 1994,11. - *in Bearbeitung*

[Hillis85] Hillis D., 1985, "The Connection Machine", MIT Press, Cambridge, MA, pp 124-126.

[Hong88] Hong J.-W., Tan X.-N., Chen M., 1988, "From local to global: An analysis of nerarest neighbor balancing on hypercube", Proc. 1988 ACM Symp. on SIGMETRICS, pp 73-82.

[Horowitz78] Horowitz E., Sahni S., 1978, "Fundamentals of computer algorithms", Computer Science Press.

[Horton93] Horton G., 1993, "A multi-level diffusion method for dynamic load balancing", Parallel Computing, vol 19, pp 209-218.

[Huang89] Huang S.-R., Davis L. S., "Parallel iterative A* search: An admissible distributed heuristic search algorithm", Proc. of the 11th Int. Joint Conf. on Artificial Intelligence, 1989, pp 23-29.

[Huang90] Huang S-R., Davis L. S., 1990, "Speedup analysis of centralized parallel heuristic search algorithms", Int. Conf. on Parallel Processing, vol 3, pp 18-21.

[Imai79b] Imai M., Fukumara T., Yoshida Y., 1979, "A parallelized branch-and-bound algorithm: Implementation and efficiency", Systems-Computers-Control, vol 10, no 3, pp 62-70.

[Janakiram88] Janakiram V. K., et al., 1988, "A randomized parallel branch-and-bound algorithm", Int. Jour. of Parallel Programming, vol 17, no 3, pp 277-301.

[Karypis92] Karypis G., Kumar V., 1992, "Unstructured tree search on SIMD parallel computers", Technical Report 92-21, Department of Computer Science, University of Minnesota, Minneapolis, April 1992.

[Kindervater89] Kindervater G. A. B., Lenstra J. K., Rinnooy Kan A. H. G., 1989, "Perspectives on parallel computing", Operations Research, vol 37, no 6, pp 985-990.

[Korf85] Korf R. E., 1985, "Depth-first iterative-deepening: An optimal admissible tree search", Artificial Intelligence, vol 27, pp 97-109.

[Korf90] Korf R. E., 1990, "Real-time heurstic search", Artificial Intelligence, vol 42, pp 189-211.

[Kumar87] Kumar V., Rao N., "Paralllel depth first search - Part II - Analysis", Int. Jour. of Parallel Programming, vol 16, no 6, 1987.

[Kumar88] Kumar V., Ramesh K., Rao V. N., 1988, "Parallel best-first search of state-space graphs: A summary of results", Proc. of the 1988 National Conf. on AI (AAAI-88).

[Kumar91] Kumar V., Ananth G. Y., Rao V. N., 1991, "Scalable load balancing techniques for parallel computers", Technical Report 91-55, Department of Computer Science, University of Minnesota.

[Lai85a] Lai T.-H., Sprague A., 1985, "Performance of parallel branch-and bound algorithms", IEEE Trans. on Computers, vol C-34, no 10, pp 962-964; MAG Lab papers, no 33.

[Lawler66] Lawler E. L., Wood D. E., 1966, "Branch-and-bound methods: A survey", Operations Research, vol 14, pp 699-719.

[Lawler89] Lawler E. L., Lentsra J. K., Rinnoy Kan A. H. G., Shmoys D. B., 1989, "Sequencing and scheduling: Algorithms and complexity", Report BS-R8909, Department of Operations Research , Statistics, and System Theory, Amsterdam, The Netherlands.

[Li84a] Li G. J., Wah B. W., 1984, "Computational efficency of parallel approximate branch-and-bound algorithms", Int. Conf. on Parallel Processing, Bellaire, Michigan, August, pp 473-480, IEEE Comp. Soc., Washington, D. C.

[Lin87] Lin F. C. H., Keller R. M., 1987, "The gradient model load balancing method", IEEE Trans. on Software Engineering, vol 13, no 1, pp 32-38.

[Lüling92] Lüling R., Monien B., "Load balancing for distributed branch-and-bound algorithm", Proc. 6th Int. Parallel Processing Symp., pp 543-548.

[Ma88] Ma R. P., Tsung F. S., Ma M. H., 1988, "A dynamic load balancer for a parallel branch-and-bound algorithm", Proc. of the 3rd Conf. on Hypercubes Concurrent, Computers, and Applications, Pasadena, CA, pp 1505-1513.

[Mahanti93] Mahanti A., Daniels C. J., 1993, "A SIMD approach to parallel heuristic search", Artificial Intelligence, vol 60, no 2, pp 243-282.

[MasPar92a] "MasPar system overview, software version 3.0", Juli 1992, PN 9300-0100.

[Miller89] Miller D. L., Pekney J. F., 1989, "Results from a parallel branch-and-bound algorithm for solving large symmetric traveling salesman problems", Operations Research Letters, vol 8, pp 129-135.

[Mohan83] Mohan J., 1983, "Experience with two parallel programs solving the traveling salesman problem", Proc. of the Int. Conf. on Parallel Processing, Bellaire, Michigan, Aug. 1983, pp 191-193, IEEE Comp. Soc., Washington, D. C.

[Müller70] Müller-Merbach H., 1970, "Operations Research", Vahlen-Verlag, Berlin, Frankfurt.

[Nilsson82] Nilsson N. J, 1982, "Principles of artificial intelligence ", Springer Verlag.

[Pargas88] Pargas R. P., Wooster E. D., 1988, "Branch-and-bound algorithms on a hypercube", Proc. of the 3rd Conf. on Hypercube, Concurrent Computers, and Applications, Pasadena, CA.

[Powley93] Powley C., Ferguson C., Korf R. E., 1993, "Depth-first heuristic search on a SIMD machine", Artificial Intelligence, vol 60, no 2, pp 199-242.

[Prechelt93] Prechelt L., 1993, "Measurements of MasPar MP-1216A communication operations", Interner Bericht Nr. 1/93, Fakultät für Informatik, Universität Karlsruhe.

[Qian91] Qian X.-S., Yang Q., 1991, "Load balancing on generalized hypercube and mesh multi-processors with LAL", Proc. 11th Int. Conf. on Distributed Computing Systems, pp 402-409.

[Quinn86] Quinn M. J., Deo N., 1986, "An upper bound for the speedup of parallel best-bound branch-and-bound algorithms", BIT, vol 26, no 1, pp 35-43.

[Rembold93] Rembold U., Nnaji B. O., Storr A., 1993, "Computer integrated manufacturing and engineering", AddisonWesley.

[Rothkopf66] Rothkopf M., 1966, "Scheduling independent tasks on parallel processors", Management Science, vol 11, no 3, pp 437-447.

[Roucairol88] Roucairol C., 1988, "Parallel branch and bound algorithms: An Overwiew", Proc. of the Int. Workshop on Parallel and Distributed Algorithms, Gers, France, pp 153-163.

[Sanders94] Sanders P., 1994, "Analysis of random polling dynamic load balancing", Interner Bericht 12/94, Fakultät für Informatik, Universität Karlsruhe.

[Schabernack92] Schabernack, 1992, "Lastausgleichsverfahren in verteilten Systemen - Überblick und Klassifikation", Informationstechnik it, vol 34, no 5, pp 280-295.

[Schneider91] Schneider H. J. (ed.), 1991, "Lexikon der Informations- und Datenverarbeitung", 3. Auflage, Oldenburg-Verlag.

[Sprague90] Sprague A. P., 1990, "Analysis of the parallel branch-and-bound algorithm on shared memory conputers", Technical Report CIS-TR-90-02, Department of Computer and Information Science, University of Alabama at Birmingham.

[Stalp93] Stalp M., 1993, "Lastverteilung für Branch-and-bound-Algorithmen auf einem SIMD-Rechner", Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe.

[Suttner93] Suttner C. B., Jobmann M. R., 1993, "Simulation analysis of static partitioning with slackness", 2nd Int. Workshop on Parallel Processing for Artificial Intelligence (PPAI-93), August 29, Chambery, France, pp 22-27.

[Taudes91] Taudes A., Netousek T., 1991, "Implementing branch-and-bound algorithms on a cluster of workstations", Eds: Grauer M., Pressmar D. B., Parallel computing and mathematical optimization, Proc., Springer.

[Vornberger86] Vornberger O., 1986, "Implementing brach-and-bound in a ring of processors", Proc. of CONPAR 86, Lecture Notes on Computer Science 237, Springer Verlag, pp 157-164.

[Vornberger87] Vornberger O., 1987, "Load balancing in a network of transputers", 2nd Int. Workshop on Distributed Algorithms, Amsterdam, pp 116-126.

[Wah82b] Wah B. W., Yu C. F., 1982, "Probabilistic modeling of branch-and-bound algorithms", Proc. COMPSAC, Nov., pp 647-653.

[Wah84b] Wah B. W., Ma Y. W. E., 1984, "MANIP – A multicomputer architecture for solving combinatorial extremum search problems", IEEE Trans. on Computers, vol C-33, no 5, pp 377-390.

[Willebeek90] Willebeek-LeMair M., Reeves A. P., 1990, "Local vs. global strategies for dynamic load balancing", Proc. Int. Conf. on Parallel Processing, vol 1, pp 569-570.

[Willebeek93] Willebeek-LeMair M. H., Reeves A., 1993, "Strategies for dynamic load balancing on highly parallel computers", IEEE Trans. on Parallel and Distributed Systems, vol 4, no 9.

[Williams91] Williams R. D., 1991, "Performance of dynamic load balancing algorithms for unstructured mesh calculations", Concurrency: Practice Experience, vol 3, no 5, pp 457-481.

[Xu93] Xu C. Z., Lau F. C. M., 1993, "Optimal parameters for load balancing using the diffusion method in k-ary n-cube networks", Information Processing Letters, vol 47, pp 181-187.

8.2 Symboltabelle

Hier folgt eine alphabetisch sortierte Liste mit den verwendeten Symbolen und Bezeichner. Die in Klammer angegebenen Kapitel geben den Ort der Einführung an, andernfalls werden die Symbole an mehreren Stellen verwendet.

A	Anzahl aktiver Prozessoren (Prozessoren mit $L_i \geq 1$) (Kapitel 2.1.2)
α	Diffusionsparameter mit $0 < \alpha < 1$ (Kapitel 2.2.1)
b	(mittlerer) Verzweigungsfaktor des betrachteten Suchbaums
$C_{i,d}(t)$	die boolesche Funktion gibt an ob zum Zeitpunkt t der Prozessor i ein Lastelement in Dimension d verschiebt (Kapitel 4.1.2)
D	Dimension des betrachteten Verbindungsnetzwerks
d	Tiefe des betrachteten Suchbaums
$\Delta(i)$	Indexmenge der zu Prozessor i direkt benachbarten Prozessoren (Kapitel 2.2.1)
$\delta_i(j)$	Lasttransfer zwischen Prozessor i und seinem direkten Nachbarn $j \in \Delta(i)$ (Kapitel 2.1.3)
D^K	dynamischer Trigger nach [Karypis92] (Kapitel 2.1.1)
D^P	dynamischer Trigger nach [Powley93] (Kapitel 2.1.1)
d_w	kleinste Tiefe des Suchbaums mit $w_{d_w} \geq P$ (Kapitel 3.2)
$f(x)$	Kriterien- bzw. Zielfunktion über die Suchbaumknoten x (Kapitel 1.2)
$g(x)$	untere Schranke für die Kosten $f(x)$ der vervollständigten Lösung des Teilproblems x (Bewertungsfunktion) (Kapitel 1.2)
I	Menge aller zulässigen Identifikatoren für die Prozessoren, Indexmenge (Kapitel 4.1.2)
k	Tiefe der k-Expansion, $k_1 = \lfloor \log_b(P) \rfloor$ und $k_2 = \lceil \log_b(P) \rceil$ (Kapitel 5.1)
K, K_i	Anzahl der Prozessoren in pro Dimension (in Dimension i)
L_{diff}	$= \max\{\lvert L_i - L_j \rvert, i, j \in I\}$, maximale Lastdifferenz (Kapitel 4.2)
L_i	Belastung des Prozessors i
$\bar{L}$	mittlere Belastung der Prozessoren
LM-C	bezeichnet das Flüssigkeitsmodell mit Shift-Bedingung C (Kapitel 4)
$L_{\max}$	$= \max\{L_i \mid i \in I\}$, maximale Belastung (Kapitel 4.2)
L_{sum}	Gesamtbelastung aller Prozessoren
m	Anzahl von Maschinen, entspricht Verzweigungsfaktor b (Kapitel 7)
$\mathbb{N}$	Menge der natürlichen Zahlen (inkl. Null)

n	Anzahl von Jobs, entspricht Suchbaumtiefe d (Kapitel 7)
$N(b, k)$	Knotenanzahl eines vollständigen Suchbaums mit Verzweigungsfaktor $b > 0$ und Tiefe k, $N(b, k) := \sum_{i=0}^{k} b^i = \frac{b^{k+1} - 1}{b - 1}$
NP	Klasse der nicht-deterministisch polynomial lösbaren Probleme
O-Kalkül:	Seien f und g Funktionen vom Typ $\mathbb{N} \to \mathbb{N}$ oder $\mathbb{R} \to \mathbb{R}$ dann gelte:
$f \in O(g)$	$:\Leftrightarrow \exists\, c > 0\colon \exists\, x_0\colon \forall\, x > x_0\colon [f(x) \le c \cdot g(x)]$ "f wächst asymptotisch höchstens so stark wie g"
$f \in \Theta(g)$	$:\Leftrightarrow f \in O(g) \wedge f \in \Omega(g)$ "f wächst asymptotisch genauso stark wie g"
$f \in \Omega(g)$	$:\Leftrightarrow g \in O(f)$ "f wächst asymptotisch mindestens so stark wie g"
P	Klasse der deterministisch polynomiell lösbaren Probleme
P	Anzahl verfügbarer Prozessoren des Parallelrechners
P^f	periodischer Trigger mit Frequenz f (Kapitel 4.4)
P_i	Prozessor mit Index $i \in I$
$\mathbb{R}$	Menge der reellen Zahlen
R	Wurzelknoten eines Suchbaums oder eines Teilbaums
σ_i	$= \sum_{j=1}^{i} L_j$, Präfix-Summe der Belastung L_j von Prozessor 1 bis Prozessor i (Kapitel 2.1.3)
S^x	statischer Trigger mit Schwellwert x (Kapitel 2.1.1)
T_{active}	Summe der Arbeitszeit aller Prozessoren seit der letzten Lastverteilungsphase (Kapitel 2.1.1)
$T_{\text{B\&B}}$	Zeit seit der letzten Lastverteilungsphase (Kapitel 2.1.1)
T_{DI}	Bearbeitungszeit der direkten Initialisierung (Kapitel 3.2.4)
T_{EDI}	Bearbeitungszeit der erweiterten direkten Initialisierung (Kapitel 3.2.5)
T_{EI}	Bearbeitungszeit der enumerativen Initialisierung (Kapitel 3.2.2)
T_{eval}	Zeit zur Berechnung der heuristischen Kostenabschätzung (untere Schranke) für einen Knoten durch die Bewertungsfunktion
T_{gen}	Zeit zum Generieren eines Nachfolgerknotens in der Tiefe 1 bzw. k mit Hilfe der Expansionsfunktion
T_{idle}	Summe der Stillstandszeit aller Prozessoren, seit der letzten Lastverteilungsphase (Kapitel 2.1.1)
T_{iter}	Zeit für eine B&B-Iteration ohne OPEN-Operationen
T_{LV}	Zeit um in der nächsten Lastverteilungsphase alle Prozessoren wieder zu beschäftigen (Abschätzung) (Kapitel 2.1.1)

T_{open}	Zeit zum Einsortieren und Entnehmen eines Knoten in die Datenstruktur OPEN
T_{red}	Zeit für einen Reduktionsschritt (Kapitel 5.3.1)
T_{RI}	Bearbeitungszeit der Wurzelinitialisierung (Kapitel 3.2.1)
T_{send}	Zeit für Versenden eines Knotens an alle Prozessoren (Broadcast)
T_{SI}	Bearbeitungszeit der selektiven Initialisierung (Kapitel 3.2.3)
T_{trans}	Zeit für Transfer eines Suchbaumknotens zwischen zwei direkt benachbarte Prozessoren
W	Gesamtzahl zu expandierender Knoten eines Suchbaums (Kapitel 7)
w_i	mittlerer Anteil unbeschnittener Knoten in der Tiefe i (Kapitel 3.2)
$\overline{w}$	mittlerer Anteil unbeschnittener Knoten des gesamten Suchbaums, Beschneidungsgrad (Kapitel 3.2)
x, y	Knoten des Suchbaums bzw. (partielle) Lösungen bzw. Teilprobleme
z	derzeit beste Lösung (incumbent)
$Z(i, j)$	Zuordnungsfunktionen von Knoten zu Prozessoren (Kapitel 3.2.5)

8.3 Schlagwortverzeichnis

Hier sind die verwendeten deutschen und englischen Fachwörter alphabetisch sortiert aufgelistet. Die erste Seitenzahl gibt die Stelle der Definition an. Zusätzliche Seitenzahlen markieren einen für das Fachwort interessanten Zusammenhang.