Iterative Tiefensuche auf einem massiv parallelen System

Universität–Gesamthochschule Paderborn
PC^2 — Paderborn Center for Parallel Computing
33095 Paderborn

Diplomarbeit
für den integrierten Studiengang Informatik
im Rahmen des Hauptstudiums HS II

von

Volker Schnecke
Vogelsberg 20
33165 Lichtenau

vorgelegt bei
Dr. Alexander Reinefeld

Paderborn, März 1994

Inhaltsverzeichnis

1 Einleitung **3**

2 Iterative Tiefensuche **6**
- 2.1 Grundlegende Definitionen 6
- 2.2 Baumsuchverfahren 7
  - 2.2.1 Allgemeine Baumsuche 7
  - 2.2.2 Tiefensuche 8
  - 2.2.3 Breitensuche 8
- 2.3 Der Algorithmus A* 8
- 2.4 Der Algorithmus IDA* 13
- 2.5 Anwendungen der iterativen Tiefensuche 16
  - 2.5.1 Das n×n–Puzzle 16
  - 2.5.2 Die Floorplan–Optimierung 18
  - 2.5.3 Das Cutting–Stock–Problem 20

3 Die Leistungsanalyse paralleler Algorithmen **21**
- 3.1 Speedup, Effizienz, Ineffizienz und Kosten 21
- 3.2 Das Gesetz von Amdahl 24
- 3.3 Der Overhead eines parallelen Algorithmus 26
- 3.4 Die Isoeffizienz-Funktion 29

4 Parallelisierungsansätze der iterativen Tiefensuche **32**
- 4.1 Die Lastverteilung in parallelen Baumsuchverfahren 32
- 4.2 MIMD–Rechner 33
  - 4.2.1 PIDA* – Parallel IDA* 33
  - 4.2.2 Ähnliche Ansätze 34
  - 4.2.3 PWS – Parallel Window Search 35
- 4.3 SIMD–Rechner 35
  - 4.3.1 SIDA* – IDA* für SIMD–Architekturen 35

5 Der Algorithmus AIDA* 38
5.1 Die Architektur des Algorithmus AIDA* 38
5.2 Die erste Phase: Synchrone Expansion 40
5.3 Die zweite Phase: Statische Lastverteilung und parallele Expansion 40
5.4 Die dritte Phase: Dynamischer Lastausgleich am Ende jeder Iteration 41
5.5 Verbesserungen von AIDA* 42

6 Die Implementierung von AIDA* 44
6.1 Der Parsytec GCel und das Betriebssystem PARIX 44
6.2 Das Prozeßmodell 45
6.3 Das Lastverteilungsverfahren 46
6.3.1 Die initiale Lastverteilung in den ersten beiden Phasen 46
6.3.2 Die dynamische Lastverteilung in der dritten Phase 47
6.4 Die Terminierung 50
6.5 Auswirkungen der unterschiedlichen Teilbaumgrößen 52
6.6 Die Ergebnisse für das Puzzle 53
6.6.1 Das 15-Puzzle 55
6.6.2 Die Overheads 58
6.6.3 Das 19-Puzzle 63
6.7 Die Floorplan-Optimierung 64

7 Die Skalierbarkeit von AIDA* 68
7.1 Die Skalierbarkeit des Algorithmus PIDA* 68
7.1.1 Die Isoeffizienz-Funktion für den Ring 68
7.1.2 Die Isoeffizienz-Funktion für den Torus 70
7.2 Die Auswirkungen der sequentiellen Phase in AIDA* 71
7.3 Die Isoeffizienz-Funktion von AIDA* 73

8 Zusammenfassung und Ausblick 76

Kapitel 1

Einleitung

Suche ist ein wichtiger Ansatz zur Problemlösung in der *künstlichen Intelligenz* und der *Operations Research*. Sie findet Anwendungen in der kombinatorischen Optimierung, der Spielprogrammierung, in planenden Systemen und der Robotics. Während der Suche wird ein Entscheidungsbaum durchlaufen und in diesem Baum nach einem Lösungsknoten gesucht. Im allgemeinen ist der Suchbaum nicht explizit gegeben, sondern wird durch einen Ausgangszustand und sich daraus ergebenden Folgezuständen definiert. Die Wurzel des Baumes repräsentiert den Ausgangszustand, die übrigen Knoten die Folgezustände. Ausgehend von der Wurzel wird der Suchbaum konstruiert und nach einem Lösungsknoten, der einen Zielzustand repräsentiert, durchsucht.

Um die Effizienz der Suche zu steigern, werden Informationen des zu lösenden Problems, sogenannte *Heuristiken*, verwendet. Diese Heuristiken kann man als „Faustregeln“ bezeichnen, welche die Suche in die günstigeren Bereiche des Baumes lenken: Jeder Knoten wird durch eine heuristische Funktion bewertet, und die Knoten mit einem günstigeren Wert werden bei der weiteren Suche bevorzugt.

In dieser Arbeit wird ein spezielles Suchverfahren, die *iterative Tiefensuche*, parallelisiert. Die diesem Verfahren zugrundeliegende Methode ist die reine Tiefensuche. Ausgehend von der Wurzel des Suchbaumes werden bei der Suche die Nachfolger eines Knotens generiert, bewertet und gespeichert. Danach wird einer der zuletzt gespeicherten Knoten untersucht und seine Nachfolger generiert. Dieses hat zur Folge, daß die Suche zunächst in die tieferen Ebenen des Baumes vordringt. Bei der *Breitensuche*, einem anderen grundlegenden Baumsuchverfahren, werden die Ebenen *(Level)* des Baumes von oben nach unten der Reihe nach durchsucht. Dieses hat den Nachteil, daß praktisch alle Knoten auf einer Ebene gespeichert werden müssen, wobei die Größe der Ebenen in der Regel exponentiell mit der Suchtiefe wächst. Der Speicherplatzbedarf bei der Tiefensuche wächst hingegen nur linear mit der Suchtiefe, was dieses Verfahren für praktische Anwendungen günstiger erscheinen läßt. Die Tiefensuche hat jedoch zwei große Nachteile: Zum einen kann, falls der Suchbaum unendlich groß (oder sehr groß) ist, die Suche unendlich in die

Tiefe gehen (bzw. sehr zweitaufwendig sein), zum anderen ist die erste gefundene Lösung nicht unbedingt optimal[1]. Um die Suche nicht unendlich in die Tiefe gehen zu lassen, kann man eine Tiefenschranke einführen. Der Baum wird dann nur bis zu einer bestimmten Tiefe durchsucht, danach wird die Suche in einer höheren Ebene, die noch nicht vollständig bearbeitet wurde, fortgeführt *(Backtracking)*. Das Problem liegt hier jedoch in der Wahl dieser Tiefenschranke: Wird sie zu klein gewählt, so kann es sein, daß kein Lösungsknoten innerhalb des durchsuchten Teilbaumes liegt, wird sie zu groß gewählt, so ist die gefundene Lösung eventuell nicht optimal.

Die *iterative Tiefensuche* umgeht die Nachteile der reinen Tiefensuche für die meisten Anwendungen und stellt die Optimalität einer gefundenen Lösung sicher. Es werden eine Reihe von beschränkten Tiefensuchen durchgeführt, wobei die Tiefenschranke jeweils um einen minimalen Wert erhöht wird. Der Baum wird somit wie bei der Breitensuche Ebene für Ebene durchsucht, wobei der Speicherplatzbedarf nur linear in der Suchtiefe wächst. Ein Nachteil der iterativen Tiefensuche ist die redundante Expansion der Knoten in den oberen Ebene des Baumes. Der Mehraufwand ist jedoch vernachlässigbar, falls die Größe der Ebenen exponentiell mit der Suchtiefe wächst.

Ein heuristischer iterativer Tiefensuchalgorithmus ist der Algorithmus *IDA**. In diesem Algorithmus werden kostenbeschränkte Tiefensuchen durchgeführt, wobei die Kostenschranke nach einer erfolglosen Iteration um einen minimalen Wert erhöht wird. Falls die Heuristik die Kosten von dem aktuellen Knoten zu einem Lösungsknoten nicht überschätzt, so findet dieser Algorithmus eine optimale Lösung. Der Algorithmus *IDA** eignet sich besonders für Probleme mit einer geringen Lösungsdichte und einer schlechten Abschätzbarkeit von oberen bzw. unteren Schranken für die Kosten einer Lösung. Trotz der Verwendung von Heuristiken sind die sich ergebenden Suchbäume noch sehr groß, so daß eine Parallelisierung dieses Verfahren lohnenswert ist.

Die Parallelverarbeitung hat in den letzten Jahren ständig an Bedeutung gewonnen. Fortschritte in der Hardware-Entwicklung haben dazu geführt, daß selbst Systeme mit mehreren hundert oder gar tausend Prozessoren heute keine Seltenheit mehr sind. Das Hauptproblem bei der Programmierung dieser *massiv parallelen Systeme* liegt in der *Lastverteilung*. Das zu lösende Problem muß in kleine Teilprobleme zerlegt werden, welche die Prozessoren des Systems weitgehend unabhängig voneinander bearbeiten. Da eine gleichmäßige Verteilung dieser Teilprobleme nur selten zu erreichen ist, muß während der Programmausführung die Last unter den einzelnen Prozessoren umverteilt werden. Diese *dynamische Lastverteilung* beeinträchtigt die Leistung eines parallelen Algorithmus, da hierfür wertvolle Rechenzeit verloren geht. Wenngleich die Lastverteilung auf allen parallelen Systemen nötig ist, so sind doch die Auswirkungen

[1] In diesem Kapitel sei die optimale Lösung diejenige, welche durch den am höchsten im Baum gelegenen Lösungsknoten repräsentiert wird.

auf massiv parallelen Systemen am größten.

Im Mittelpunkt dieser Arbeit steht der Algorithmus *AIDA* (Asynchronous parallel IDA*)*, eine parallele Version des heuristischen iterativen Tiefensuchalgorithmus *IDA**. Als Anwendung wurde das *n×n–Puzzle* gewählt, welches ein beliebtes Suchproblem aus der künstlichen Intelligenz ist. Zudem wird die Anwendbarkeit dieses Verfahrens auf die *Floorplan-Optimierung*, einem Problem aus dem *VLSI*-Entwurf, untersucht. Der Algorithmus *AIDA** wurde auf einem massiv parallelen System mit 1024 Prozessoren implementiert. Bei der Parallelisierung stand die *Skalierbarkeit* des parallelen Algorithmus im Vordergrund. Ein paralleler Algorithmus ist skalierbar, falls für die gleichen Probleme auf beliebig großen Systemen annähernd die gleiche *Effizienz* erreicht wird. Der Grund für die schlechte Skalierbarkeit vieler paralleler Algorithmen liegt in dem großen *Overhead* dieser Ansätze. Der Overhead beschreibt den Mehraufwand der Parallelisierung und wird maßgeblich durch die Kosten für die Lastverteilung bestimmt. Der Gesamtoverhead wächst im allgemeinen mit der Zahl der Prozessoren. Da die Rechenzeiten auf Systemen mit mehr als eintausend Prozessoren entsprechend kurz werden, wird der Anteil des Overheads an der Laufzeit sehr groß, so daß die Auswirkungen auf die Effizienz stärker sind als in kleineren parallelen Systemen.

Diese Arbeit gliedert sich wie folgt: Nach einer Vorstellung der sequentiellen Baumsuchalgorithmen und einiger Anwendungen der iterativen Tiefensuche im zweiten Kapitel werden im Kapitel 3 die Grundkonzepte der Leistungsanalyse von parallelen Algorithmen vorgestellt. In Kapitel 4 werden einige andere Parallelisierungsansätze beschrieben, im nächsten Kapitel folgt die Beschreibung des Algorithmus *AIDA**. Die Ergebnisse der Implementierung des Algorithmus für das n×n–Puzzle und die Floorplan-Optimierung werden in Kapitel 6 aufgeführt. In Kapitel 7 wird die bessere Skalierbarkeit von *AIDA** gegenüber eines weit verbreiteten Parallelisierungsansatzes nachgewiesen.

Kapitel 2

Iterative Tiefensuche

In diesem Kapitel werden *Baumsuchverfahren* beschrieben und der Zusammenhang zwischen einem *Optimierungsproblem* und der Baumsuche dargestellt. Nach der Beschreibung der allgemeinen Baumsuchverfahren werden der heuristische Baumsuchalgorithmus *A** und der iterative Tiefensuchalgorithmus *IDA** vorgestellt. Als praktische Beispiele für Optimierungsprobleme, die sich mittels heuristischer iterativer Tiefensuche lösen lassen, werden das $n \times n$*-Puzzle*, die *Floorplan-Optimierung* und das *Cutting-Stock-Problem* beschrieben.

2.1 Grundlegende Definitionen

Definition 2.1 (Baum) *Ein* **Baum** $\mathcal{B} = (\mathcal{V}, \mathcal{E}, r)$ *wird durch eine Knotenmenge* $\mathcal{V}$*, eine Kantenmenge* $\mathcal{E} \subseteq (\mathcal{V} \times \mathcal{V})$ *und einen ausgezeichneten Knoten* $r \in \mathcal{V}$*, der* **Wurzel***, beschrieben. Eine Kante* $e = (n_1, n_2)$ *ist von einem Knoten* n_1 *auf dessen* **Nachfolger***, den Knoten* n_2 *gerichtet.* n_1 *heißt* **Vorgänger** *von* n_2*. Die Wurzel* r *ist der einzige Knoten im Baum ohne Vorgänger, alle anderen Knoten haben genau einen Vorgänger. Knoten ohne Nachfolger werden als* **Blätter** *des Baumes bezeichnet, die übrigen Knoten sind die* **inneren Knoten***.*

Die Bäume werden anhand der maximalen Zahl der Nachfolger ihrer Knoten in Klassen eingeteilt. So werden etwa als *binäre Bäume* diejenigen Bäume bezeichnet, deren Knoten maximal zwei Nachfolger haben. Als *Verzweigungsfaktor* eines Baumes wird die durchschnittliche Zahl der Nachfolger eines inneren Knotens beschrieben.

Definition 2.2 (Tiefe) *Die* **Tiefe** *eines Knotens* $n \in \mathcal{V}$ *in einem Baum* $\mathcal{B} = (\mathcal{V}, \mathcal{E}, r)$ *beschreibt die Länge (d. h. die Zahl der Kanten) des Pfades von der Wurzel* r *zum Knoten* n*. Die Wurzel* r *hat die Tiefe 0. Die Tiefe des Nachfolgers eines Knotens mit der Tiefe* d *beträgt* $d+1$*.*

Die Bäume, die durch die im folgenden vorgestellten Algorithmen durchsucht werden, repräsentieren *Optimierungsprobleme*. Die Knoten der Bäume beschreiben *Zustände* (Konfigurationen) eines Systems, die Wurzel repräsentiert den *Ausgangszustand*. Jede Kante beschreibt einen möglichen Übergang von einem Zustand zu einem erlaubten Folgezustand, welcher durch

den entsprechenden Nachfolger repräsentiert wird. Der Baum wird nach einem *Lösungsknoten* durchsucht, welcher einen *Zielzustand* des Systems darstellt. Gesucht ist im Rahmen des Optimierungsproblems ein kostengünstiger Lösungsknoten, wobei die Kosten durch den Knoten oder durch die Kosten des Pfades *(Lösungspfad)* von der Wurzel zu diesem Knoten beschrieben werden.

2.2 Baumsuchverfahren

Man kann die Baumsuchverfahren in zwei Gruppen einteilen:

- Bei der *uninformierten Suche* wird der Suchbaum in beliebiger Reihenfolge durchlaufen und ein Lösungsknoten gesucht.
- Bei der *heuristischen* oder *gerichteten Suche*, werden in jedem Knoten die Nachfolger anhand einer Bewertungsfunktion *(Heuristik)* bewertet und zunächst die „besten“ Knoten bei der weiteren Suche berücksichtigt.

2.2.1 Allgemeine Baumsuche

```
Program Treesearch;
  OPEN := ∅;
  insert r into OPEN;
  while OPEN <> ∅ do
      retrieve node n from OPEN;                                    (*)
      if n is goal-node then output (solution) and exit;
      expand n and insert all successors n.i into OPEN;
  output ('no solution found');
```

Abbildung 2.1: Der allgemeine Baumsuchalgorithmus

Die Grundstruktur der im folgenden dargestellten Baumsuchalgorithmen ist identisch (siehe Abbildung 2.1). Es existiert eine Knotenliste `OPEN`, welche die *Suchfront* enthält. Die Suchfront ist die Menge aller Knoten, deren Nachfolger während der Suche noch nicht expandiert wurden. Als *Expansion* eines Knotens wird die Generierung und Bewertung der Nachfolger dieses Knotens bezeichnet. Zu Beginn der Suche besteht die `OPEN`-Liste nur aus der Wurzel r des Suchbaumes. Die Auswahl des Knotens, der in Zeile $(*)$ zur Expansion aus der `OPEN`-Liste entnommen wird, ist entscheidend für den Verlauf der Suche, d. h. den Weg, auf dem der Baum bei der Suche durchlaufen wird. Hier gibt es zwei verschiedene Verfahren:

- Tiefensuche *(Depth-First-Search)*
- Breitensuche *(Breadth-First-Search)*

2.2.2 Tiefensuche

Bei der *Tiefensuche* (Depth-First-Search, DFS) wird die `OPEN`-Liste als LIFO- *(Last-In-First-Out-)* Liste verwaltet. Dabei wird der zuletzt in die Liste eingefügte Knoten als erster wieder entnommen und dessen Nachfolger generiert. Dies hat zur Folge, daß niemals ein Knoten mit Tiefe d aus der `OPEN`-Liste entnommen wird, solange noch Knoten mit einer Tiefe $> d$ in der Liste enthalten sind.

Damit die Suche nicht endlos auf erfolglosen Pfaden (d. h. Pfade, die nicht Teil eines Lösungspfades sind) in die Tiefe geht, ist es sinnvoll, eine Tiefenschranke einzuführen. Liegt ein Knoten im Suchbaum unterhalb dieser Tiefenschranke, so wird die Suche an dieser Stelle abgebrochen, d. h. seine Nachfolger werden nicht in die `OPEN`-Liste aufgenommen.

Die Speicherplatzkomplexität der Tiefensuche mit Tiefenschranke d beträgt $O(d)$, da immer nur die Knoten auf einem Pfad der Länge d und deren direkte Nachfolger in der `OPEN`-Liste gehalten werden müssen. Die Zeitkomplexität der Tiefensuche beträgt $O(b^d)$, wobei b der Verzweigungsfaktor des Suchbaumes ist, da bei der Tiefensuche maximal $1+b+b^2+b^3+\ldots+b^d = O(b^d)$ Knoten betrachtet werden

2.2.3 Breitensuche

Bei der *Breitensuche* (Breadth-First-Search, BFS) ist die `OPEN`-Liste eine FIFO- *(First-In-First-Out-)* Liste. Die Knoten werden dabei in der Reihenfolge untersucht, in der sie in die `OPEN`-Liste eingefügt worden sind. Falls in einem Suchbaum mehrere Lösungsknoten existieren, so wird bei der Breitensuche – im Gegensatz zur Tiefensuche – ein kürzester Lösungspfad gefunden. Ein Knoten mit Tiefe d im Suchbaum wird erst expandiert, wenn kein Knoten mit einer Tiefe $< d$ in der `OPEN`-Liste enthalten ist. Dies hat jedoch zur Folge, daß bei einer Breitensuche bis zur Tiefe d im Suchbaum die `OPEN`-Liste alle Knoten der Tiefe $d-1$ aufnehmen muß. Die Speicherplatzkomplexität eines Breitensuchalgorithmus beträgt $O(b^d)$, wobei b den Verzweigungsfaktor des Suchbaumes und d die Länge eines kürzesten Lösungspfades im Suchbaum beschreibt. Die Zeitkomplexität beträgt ebenfalls $O(b^d)$, da maximal der gesamte Baum bis zur Tiefe d durchsucht werden muß.

2.3 Der Algorithmus A*

Die Suchstrategien DFS und BFS finden – falls ein Lösungsknoten im Suchbaum enthalten ist – einen Lösungspfad, sie expandieren dabei allerdings zuviele Knoten, da sie sich nicht auf diejeni-

gen Pfade konzentrieren, die mit großer Wahrscheinlichkeit zu einem Lösungsknoten führen. Da es in der Praxis für solche Suchen Grenzen im Bezug auf die zur Verfügung stehenden Ressourcen wie Zeit und Speicherplatz gibt, müssen Alternativen zu diesen „blinden" Suchstrategien gefunden werden. Solche alternativen Strategien benutzen spezielle Informationen – sogenannte *heuristische Informationen* – des gegebenen Problems. Diese auch als *Best-First-Suchstrategien* bezeichneten Verfahren entscheiden anhand der *Heuristik*, d. h. anhand einer *Knotenbewertungsfunktion*, welcher Knoten des Suchbaumes als nächstes expandiert wird.

Ein heuristischer Baumsuchalgorithmus ist der Algorithmus A^* [HartNilssonRaphael68, Nilsson80]. In diesem Algorithmus besteht die Bewertungsfunktion $f(n)$ für einen Knoten n im Suchbaum aus zwei Teilen:

$$f(n) = g(n) + h(n)$$

Die Funktion $g(n)$ beschreibt die Kosten von der Wurzel zum Knoten n, und der Funktionswert $h(n)$ sind die geschätzten (heuristischen) Kosten von diesem Knoten zu einem Lösungsknoten. Mit Hilfe der Funktion $f(n)$ läßt sich die Güte eines Pfades durch einen Knoten n abschätzen. In dem Algorithmus A^* sind die Knoten in der `OPEN`-Liste nach den Werten der Funktion $f(n)$ sortiert, so daß jeweils der Knoten mit optimalen (d. h. minimalen oder maximalen, je nach Optimierungsproblem) $f(n)$ als nächstes aus der Liste entfernt wird, dessen Nachfolger generiert und in die Liste eingefügt werden.

Weiter existiert im Algorithmus A^* neben der `OPEN`-Liste noch eine Liste `CLOSED`, welche die schon betrachteten Knoten (Zustände) enthält. Wird ein neuer Knoten n generiert, so werden beide Listen nach diesem Knoten durchsucht. Hier können drei Fälle unterschieden werden:

1. Der Knoten ist schon in der `OPEN`-Liste enthalten. In diesem Fall wird der Wert $f(n)$ des Knotens n mit dem Wert $f(n')$ seines Duplikaten n' in der Liste verglichen. Gilt $f(n') > f(n)$, so wird der Wert des Knoten in der Liste auf $f(n)$ aktualisiert[1].

2. Der Knoten ist schon in der `CLOSED`-Liste enthalten. In diesem Fall wird der Knoten mit dem minimalen Funktionswert in die `OPEN`-Liste übertragen.

3. Der Knoten ist in keiner der beiden Listen enthalten. In diesem Fall wird $(n, f(n))$ in die `OPEN`-Liste aufgenommen.

Damit der Algorithmus A^* korrekt arbeitet, muß die heuristische Funktion $h(n)$ eine bestimmte Eigenschaft erfüllen, sie muß *zulässig* sein.

[1] Bei der Formulierung des Algorithmus wird davon ausgegangen, daß ein Knoten n_1 mit Wert $f(n_1)$ günstiger ist als ein Knoten n_2 mit $f(n_2) > f(n_1)$, d. h. ein optimaler Lösungsknoten ist ein Knoten mit minimalem Wert $f(n)$.

Program $A*$;

```
OPEN := CLOSED := ∅;
insert (r, f(r)) into OPEN;
while OPEN <> ∅ do
  retrieve (n, f(n)) from OPEN with (∀(n', f(n')) ∈ OPEN : f(n) ≤ f(n'));
  if n is goal-node then output (solution) and exit;
  for all successors n.i of n do
      calculate val := f(n.i);
      if n.i in OPEN then
          retrieve (n.i, f(n.i)) from OPEN;
          insert (n.i, min(val, f(n.i))) into OPEN;
      else if n.i in CLOSED then
              retrieve (n.i, f(n.i)) from CLOSED;
              insert (n.i, min(val, f(n.i))) into OPEN;
          else insert (n.i, val) into OPEN;
output ('no solution found');
```

Abbildung 2.2: Der Algorithmus $A*$

Definition 2.3 (Zulässigkeit einer heuristischen Funktion) *Sei $h(n)$ eine heuristische Funktion, welche die Kosten des Pfades von einem Knoten n im Suchbaum zu einem Lösungsknoten abschätzt, seien $h^*(n)$ die exakten Kosten des optimalen Pfades vom Knoten n zu einem Lösungsknoten. Die heuristische Funktion $h(n)$ heißt* **zulässig**, *wenn sie die Kosten des optimalen Pfades zu einem Lösungsknoten niemals überschätzt, d. h. wenn gilt:*

$$\forall n \in \mathcal{V}: \quad h(n) \leq h^*(n)$$

Ist die heuristische Funktion in A^* nicht zulässig, so kann die Optimalität einer gefundenen Lösung nicht garantiert werden, da in diesem Fall nicht notwendigerweise ein günstigster Lösungspfad gefunden wird. Aus der Zulässigkeit der heuristischen Funktion $h(n)$ folgt also die Korrektheit des Algorithmus A^*. Dieses läßt sich durch einen Widerspruchsbeweis zeigen, in dem man die Zulässigkeit der heuristischen Funktion voraussetzt und annimmt, daß der Algorithmus A^* nicht korrekt arbeitet.

Voraussetzung:	Es existiert im Baum ein optimaler Lösungspfad mit endlicher Länge. Die heuristische Funktion h ist zulässig.
Annahme:	Der Algorithmus A^* arbeitet nicht korrekt.

Beweis durch Widerspruch:
Wenn der Algorithmus A^* nicht korrekt arbeitet, so kann man die folgenden drei Fälle unterscheiden:

(1) A^* terminiert nicht

(2) A^* terminiert, es wurde jedoch kein Lösungsknoten gefunden

(3) A^* findet einen Lösungspfad, der nicht optimal ist

(1) Falls A^* nicht terminiert, so dürfte die `OPEN`-Liste niemals leer werden. Es gilt jedoch für jeden Knoten im Suchbaum und somit für jeden Knoten in der `OPEN`-Liste:

$$\begin{aligned} f(n) &= g(n) + h(n) \\ &\geq g(n) \qquad\qquad \text{da } h(n) \geq 0 \\ &\geq g^*(n) \end{aligned}$$

Die Funktion $g^*(n)$ beschreibt die minimalen Kosten von der Wurzel r zum Knoten n. Sei $l^*(n)$ die Länge des günstigsten Pfades von der Wurzel zu diesem Knoten und sei w das minimale Kantengewicht (d. h. die Kosten der billigsten Kante) auf diesem Weg, so gilt:

$$f(n) \geq g^*(n) \geq w \cdot l^*(n)$$

Falls A^* nicht terminiert, so würden die Pfadlängen $l^*(n)$ und somit auch die Funktionswerte $f(n)$ für jeden Knoten n in der `OPEN`-Liste beliebig groß werden. Dies kann jedoch nicht sein, da man zeigen kann, daß in der `OPEN`-Liste immer ein Knoten enthalten sein muß, der auf einem optimalen Lösungspfad liegt.

Sei $(r, n_1, n_2, n_3, \ldots, n_s)$ ein optimaler Lösungspfad von der Wurzel r zu einem Lösungsknoten n_s. Sei n' derjenige Knoten mit dem kleinsten Index auf diesem Pfad, der in der `OPEN`-Liste enthalten ist. Es muß ein Knoten aus dem Lösungspfad in dieser Liste enthalten sein, da zu Beginn die Wurzel r in der `OPEN`-Liste ist und A^* terminiert, falls der Knoten n_s expandiert wird. Da alle Vorgänger des Knotens n' in der `CLOSED`-Liste sind und n' auf einem optimalen Lösungspfad liegt, hat A^* bereits einen optimalen Pfad bis zu diesem Knoten gefunden, d. h. es gilt:

$$\begin{aligned} g(n') &= g^*(n') \\ \Longrightarrow f(n') &= g(n') + h(n') \\ &= g^*(n') + h(n') \\ &\leq g^*(n') + h^*(n') \qquad \text{wg. Zulässigkeit von } h \\ &= f^*(n') \end{aligned}$$

Da $f^*(n')$ die exakten Kosten eines optimalen Lösungspfades beschreibt, der durch den Knoten n' läuft, gilt

$$f(n') = g^*(n') + h(n') \leq f^*(n') = f^*(r) \tag{2.1}$$

da für alle Knoten n_i auf dem optimalen Lösungspfad gelten muß:

$$f^*(n_i) = f^*(r)$$

Somit muß immer ein Knoten n_i, der auf einem optimalen Lösungspfad liegt, in der `OPEN`-Liste enthalten sein, also kann der Wert $f(n_i)$ für diesen Knoten nicht beliebig wachsen, da gelten muß

$$f(n_i) \leq f^*(r)$$

Daraus folgt, daß der Algorithmus A^* terminiert, falls ein endlicher Lösungspfad existiert.

(2) Nachdem nun die Terminierung des Algorithmus A^* sichergestellt ist, muß gezeigt werden, daß auch ein Lösungsknoten gefunden wird. A^* würde terminieren, wenn die `OPEN`-Liste leer ist. In diesem Fall müßte aber zuvor ein Lösungsknoten entnommen worden sein, da oben gezeigt wurde, daß immer ein Knoten auf einem Lösungspfad in dieser Liste enthalten ist. Würde A^* diesen Lösungsknoten aus der Liste entnehmen, so würde der Algorithmus terminieren. Somit ist sichergestellt, daß A^* einen Lösungsknoten findet.

(3) Damit A^* korrekt arbeitet, muß noch sichergestellt werden, daß zu dem Lösungsknoten auch der optimale Pfad gefunden wird. Hat A^* mit einer nicht optimalen Lösung terminiert (sei n der gefundene Lösungsknoten), so gilt:

$$f(n) = g(n) + h(n) = g(n) > f^*(r)$$

$f^*(r)$ sind die Kosten eines optimalen Lösungspfades. Nach Ungleichung (2.1) gilt jedoch, daß zu jedem Zeitpunkt vor der Terminierung ein Knoten n' in der `OPEN`-Liste enthalten ist, der auf einem optimalen Lösungspfad liegt, was bedeutet:

$$f(n') \leq f^*(r) < f(n)$$

Da A^* immer den Knoten mit minimalen Wert $f(n)$ aus der `OPEN`-Liste entfernt, wäre der Knoten n', der auf einem optimalen Lösungspfad liegt, vor dem Knoten n aus der Liste entfernt und expandiert worden. Dies bedeutet, daß der optimale Lösungspfad vor der Expansion des Knotens n mit $f(n) > f^*(r)$ gefunden worden wäre.

Da in Ungleichung (2.1) die Zulässigkeit der heuristischen Funktion eingeht, arbeitet der Algorithmus *A** korrekt, d. h. er terminiert mit einem optimalen Lösungspfad, falls die heuristische Funktion $h(n)$ zulässig ist.

2.4 Der Algorithmus IDA*

Aufgrund des hohen Speicherplatzbedarfs ist die praktische Anwendbarkeit des Algorithmus *A** für die meisten Probleme nicht gegeben. Die Listen `OPEN` und `CLOSED` halten jeweils die gesamte Suchfront. Dies führt im allgemeinen sehr bald zum Speicherüberlauf, außerdem ist die Größe der Suchfront nicht *a priori* abschätzbar. Es existieren zwar auf dem Algorithmus *A** basierende Algorithmen (z. B. [SarkarChakrabartiGhose92]), die mit begrenztem Speicher arbeiten, doch die Anwendung dieser Verfahren auf Probleme wie z. B. für das $n \times n - Puzzle$, welches in Kapitel 2.5.1 beschrieben wird, läßt aufgrund der großen Suchbäume keine akzeptablen Ergebnisse erwarten.

Der von Korf [Korf85] vorgestellte heuristische iterative Tiefensuch-Algorithmus *IDA** *(Iterative-Deepening A*)* vereint die Vorteile des heuristischen Suchalgorithmus *A** mit der geringen Speicherplatzkomplexität der Tiefensuche. Im Algorithmus *IDA** werden nacheinander beschränkte Tiefensuchen ausgeführt. Dabei wird die Tiefensuche durch eine Kostenschranke beschränkt, welche am Ende einer Iteration auf den minimalen Wert der Bewertungsfunktion gesetzt wird, der in dieser Iteration die Schranke überschritten hat.

Als Kostenschranke `threshold` wird initial die Bewertungsfunktion der Wurzel r des Suchbaumes gewählt:

$$\texttt{threshold} = f(r) = g(r) + h(r) = h(r)$$

Es werden somit in der ersten Iteration nur die Nachfolger $n.i$ eines Knotens n expandiert, deren Wert $f(n)$ kleiner oder gleich dem Wert $f(r)$ der Wurzel ist. Wurde in einer Iteration keine Lösung gefunden, so wird die Kostenschranke `threshold` auf das Minimum der Werte erhöht, die diese Schranke überschritten haben.

Da der Initialwert der Kostenschranke der Wert $h(r)$ ist, wird in der ersten Iteration keine optimale Lösung übergangen, wenn die heuristische Funktion zulässig ist. Da die Kostenschranke in den folgenden Iterationen auf den Minimalwert der Bewertungsfunktion aller betrachteten Knoten gesetzt wird, welche die Kostenschranke überschritten haben, ist sichergestellt, daß die erste gefundene Lösung gleichzeitig auch eine optimale Lösung ist. Die letzte Iteration, d. h. diejenige Iteration, in der die Lösung gefunden wird, wird im folgenden als *Lösungsiteration* bezeichnet. Ein Nachteil des Algorithmus *IDA** ist die redundante Expansion der Knoten in den zahlreichen Iterationen, da ein generierter Knoten nicht gespeichert wird.

```
Program IDA*;
  global var  threshold;

    function DFS(n)
      if f(n) > threshold
        return (f(n));
      if h(n) = 0
        output (solution) and exit;
      for all successors n.i of n do
        new_threshold.i := DFS (n.i);
      return ( min {new_threshold.i} );

  threshold := h(r);
  while TRUE do
    threshold := DFS (r);
```

Abbildung 2.3: Der Algorithmus *IDA** (Iterative Deepening A*)

Definition 2.4 (heuristischer Verzweigungsfaktor)
Der **heuristische Verzweigungsfaktor** *des Algorithmus* IDA* *ist das Verhältnis der Anzahl der Knoten, die in einer Iteration (außer der Lösungsiteration) expandiert werden, zur Zahl der Knoten, die in der vorherigen Iteration expandiert wurden.*

Ist der heuristische Verzweigungsfaktor für alle Iterationen größer als 1, so wächst die Zahl der in einer Iteration betrachteten Knoten exponentiell mit der Zahl der Iterationen. Sei b der heuristische Verzweigungsfaktor und d die Zahl der Iterationen. In der Lösungsiteration beträgt die Zahl der Knotenexpansionen maximal b^d. Die b^{d-1} Knoten der (d-1)-ten Iteration werden zweimal expandiert[2], nämlich in der vorletzten Iteration und in der Lösungsiteration, die b^{d-2} Knoten der (d-2)-ten Iteration werden dreimal expandiert, u. s. w. Daraus ergibt sich die Summe der vom Algorithmus *IDA** in allen Iterationen betrachteten Knoten W_{IDA^*}:

$$\begin{aligned} W_{IDA^*} &= b^d + 2 \cdot b^{d-1} + 3 \cdot b^{d-2} + \ldots + d \cdot b \\ &= b^d \cdot (1 + \frac{2}{b} + \frac{3}{b^2} + \ldots + \frac{d}{b^{d-1}}) \\ \Longrightarrow \quad W_{IDA^*} &= b^d \cdot (1 + 2 \cdot x + 3 \cdot x + \ldots + d \cdot x^{d-1}) \qquad \text{mit } x := \frac{1}{b} \end{aligned}$$

Diese Summe ist kleiner als die unendliche Reihe

$$b^d \cdot (1 + 2 \cdot x + 3 \cdot x^2 + 4 \cdot x^3 + \ldots)$$

[2]Natürlich werden diejenigen Knoten, welche höher im Suchbaum liegen mehr als zweimal expandiert, das wird im folgenden noch berücksichtigt.

welche gegen

$$b^d \cdot (1-x)^{-2}$$

konvergiert, falls $|x| < 1$ gilt.
Die Zahl der vom Algorithmus A^* in diesem Fall betrachteten Knoten beträgt

$$W_{A^*} = b^d$$

da er die gleichen Knoten wie der Algorithmus *IDA** in der Lösungsiteration betrachtet. Somit beträgt der relative Mehraufwand des Algorithmus *IDA**

$$\frac{W_{IDA^*}}{W_{A^*}} = (1 - \frac{1}{b})^{-2} = \frac{b^2}{(b-1)^2}$$

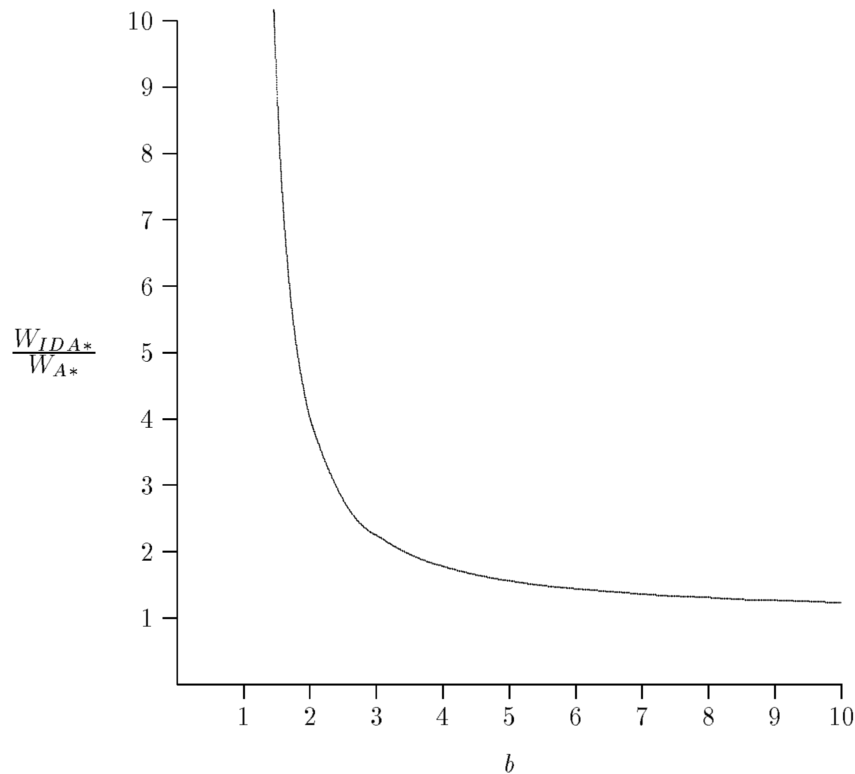


Abbildung 2.4: Der relative Mehraufwand des Algorithmus *IDA** im Gegensatz zum Algorithmus A^* bei heuristischem Verzweigungsfaktor b

Wie man in Abbildung 2.4 sieht, ist für größere heuristische Verzweigungsfaktoren b der Mehraufwand sehr gering, d. h. die Zahl der vom Algorithmus *IDA** expandierten Knoten ist asymptotisch gleich der Anzahl der vom Algorithmus A^* betrachteten Knoten [Korf88]. Aus der Zeit-Optimalität des Algorithmus A^* für exponentiell wachsende Suchbäume folgt daher

die Zeit-Optimalität des Algorithmus *IDA**. Weiter ist *IDA** auch Speicherplatzoptimal, da – wie bei der Tiefensuche – zu jedem Zeitpunkt nur der aktuelle Suchpfad gespeichert werden muß.

Der Algorithmus *IDA** eignet sich besonders für Probleme mit

- geringer Lösungsdichte, d. h. es existieren im Suchbaum nur wenige (optimale) Lösungen
- hohem heuristischen Verzweigungsfaktor
- schlechter Abschätzbarkeit von Grenzen für optimale Lösungen

In Problemen mit diesen Eigenschaften versagen die einfachen, uninformierten Suchstrategien, da die Suchbäume zu groß sind. Es können nur heuristische Suchverfahren verwendet werden, wobei *A** wegen des hohen Speicherbedarfs in der Praxis nicht geeignet ist. Auch *Branch & Bound* Ansätze versagen, da genauere Grenzwerte für die Lösungen schwer zu berechnen sind und die Lösungsdichte zu gering ist. *IDA** ist der einzige bekannte Algorithmus, der in akzeptabler Zeit optimale Lösungen für das 15-Puzzle (siehe Kapitel 2.5.1) findet.

Aufgrund der Größe der Suchbäume ist gerade auf dem Gebiet der Baumsuche eine Parallelisierung angebracht. Der Einsatz von massiv parallelen Systemen auf diesem Gebiet ist ebenfalls lohnend, da die Baumsuchverfahren eine sehr feingranulare Parallelität beinhalten und Teile des Suchbaumes weitgehend unabhängig voneinander durchsucht werden können.

2.5 Anwendungen der iterativen Tiefensuche

Die iterative Tiefensuche findet zahlreiche Anwendungen in der *Operations Research* oder in der *Künstlichen Intelligenz*. So lassen sich mit diesem Verfahren etwa Logistik-Anwendungen und Pack-Probleme lösen. Drei bekannte Anwendungen sollen im folgenden stellvertretend für eine Vielzahl von Problemen genauer definiert werden: Das $n \times n$*–Puzzle*, mit dem auch die in Kapitel 6 beschriebene Implementierung getestet wurde, die *Floorplan–Optimierung* und das *Cutting–Stock–Problem*.

2.5.1 Das n×n–Puzzle

Das $n{\times}n$*–Puzzle* (auch als (n^2-1)–Puzzle bezeichnet) besteht aus n^2-1 von 1 bis n^2-1 durchnumerierten Spielsteinen in einem festen $n{\times}n$ Rahmen. Abbildung 2.5 zeigt einige Beispielpuzzles. Die dargestellten Stellungen sind die Zielstellungen der entsprechenden Puzzles. Ein Feld im n×n–Puzzle ist leer. Auf dieses Feld kann in einem Zug ein dazu benachbarter Spielstein „geschoben“ werden. Als Lösung des Puzzleproblems gilt im folgenden die kürzeste Zugfolge, durch

	1	2
3	4	5
6	7	8

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Abbildung 2.5: Das 3×3- bzw. 8–Puzzle (links), das 4×4- bzw. 15–Puzzle (Mitte) und das 5×5- bzw. 24–Puzzle (rechts)

die eine gegebene beliebige Ausgangsstellung in die Zielstellung des n×n–Puzzles überführt werden kann.

Die Wurzel des Suchbaumes repräsentiert die Ausgangsstellung. Die Nachfolger eines Knotens repräsentieren diejenigen Stellungen, die aus der diesem Knoten entsprechenden Stellung durch Verschieben der Spielsteine entstehen. Gesucht ist somit der kürzeste Pfad von der Wurzel des Suchbaumes zu einem Lösungsknoten, d. h. einem Knoten, der die Zielstellung des n×n–Puzzles repräsentiert.

Die Bewertungsfunktion $f(n)$ für einen Knoten n setzt sich somit zusammen aus der Funktion $g(n)$, welche die Länge des Pfades von der Wurzel zum Knoten n beschreibt und der heuristischen Funktion $h(n)$. Eine mögliche Heuristik für das Puzzleproblem ist die *Manhattan-Distanz.*

Definition 2.5 (Manhattan-Distanz) *Die* **Manhattan-Distanz** *einer Stellung des n×n-Puzzle ist die Summe der kürzesten Entfernungen der fehlgestellten Spielsteine von ihrem Zielfeld. Sei* (i_x, j_x) *die Position des Spielsteines* x *in einer Stellung, wobei* i_x *die Zeile und* j_x *die Spalte angeben. Die Position des Spielsteines* x *in der Zielstellung ist* $(x\ DIV\ n, x\ MOD\ n)$. *Die Manhattan-Distanz einer Stellung des n×n-Puzzles berechnet sich somit wie folgt:*

$$MD = \sum_{x=1}^{n^2-1} (|i_x - x\ DIV\ n| + |j_x - x\ MOD\ n|)$$

Als Heuristik für das n×n–Puzzle ist die Manhattan-Distanz wegen ihrer Einfachheit und Zulässigkeit sehr verbreitet. Sie liefert allerdings nur eine sehr ungenaue Abschätzung für die Länge eines kürzesten Lösungspfades. Während der mittlere Fehler, d. h. die Differenz der Manhattan-Distanz der Ausgangsstellung und der Länge des kürzesten Lösungspfades beim 8–Puzzle 8 ist [Reinefeld93], so beträgt der Fehler für die 100 Instanzen des 15-Puzzles aus [Korf85] etwa 16. In *IDA** werden somit im Schnitt 8 Iterationen ausgeführt, bis eine Lösung

gefunden wird. Für das 19–Puzzle (eine nicht quadratische Variante des n×n–Puzzles) müssen im Durchschnitt 14 Iterationen durchgeführt werden, da der Fehler hier über 28 liegt[3].

2.5.2 Die Floorplan–Optimierung

Die *Floorplan–Optimierung* [Lengauer90] ist eine Arbeitsphase bei dem Entwurf von integrierten bzw. hochintegrierten Schaltkreisen. Diese Schaltkreise bestehen aus mehreren Teilschaltkreisen, den *Blöcken*, deren Position durch das *Layout* des Schaltkreises vorgegeben ist. Für die Größe, d. h. die Abmessungen eines Blockes gibt es mehrere Alternativen. Das Problem bei der Floorplan–Optimierung besteht darin, eine optimale Konfiguration für den Schaltkreis zu finden, wobei das Optimum eine Konfiguration mit minimaler Layout-Fläche ist. Während in der Praxis auch L-förmige Blöcke auftauchen, werden im folgenden nur Layouts mit rechteckigen Blöcken betrachtet.

Die Eingabe bei der Floorplan–Optimierung besteht aus dem Layout des Schaltkreises, welches die relative Lage der Blöcke beschreibt und die möglichen Kombinationen der Seitenlängen jedes Blockes.

Das Layout wird durch zwei gerichtete Graphen $\mathcal{G} = (\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G})$ und $\mathcal{H} = (\mathcal{V}_\mathcal{H}, \mathcal{E}_\mathcal{H})$ beschrieben. Die Knotenmengen der Graphen beschreiben die vertikalen und horizontalen Schnitte, d. h. diejenigen Linien im Layout, an denen zwei Blöcke aneinanderliegen. Die Kantenmengen repräsentieren Blöcke, deren Wände an dem entsprechenden Knoten beteiligt sind:

$$\begin{aligned}(u,v) \in \mathcal{E}_\mathcal{G} \quad &:\Longleftrightarrow \quad \exists B_i : \text{linke Kante von Block } B_i \text{ an Schnitt } u\\ &\qquad\qquad \text{und rechte Kante von } B_i \text{ an Schnitt } v\\ (w,x) \in \mathcal{E}_\mathcal{H} \quad &:\Longleftrightarrow \quad \exists B_i : \text{obere Kante von Block } B_i \text{ an Schnitt } w\\ &\qquad\qquad \text{und untere Kante von } B_i \text{ an Schnitt } x\end{aligned}$$

In Abbildung 2.6 ist ein Floorplan mit den entsprechenden Graphen $\mathcal{G}$ und $\mathcal{H}$ dargestellt.

Den Kanten der Graphen $\mathcal{G}$ und $\mathcal{H}$ werden bei der Lösung der Floorplan-Optimierung die Höhe bzw. Breite der entsprechenden Blöcke als Kantengewichte zugeordnet. Da die Lösung eines Floorplan–Optimierung–Problems der Floorplan *(Konfiguration)* mit der minimalen Fläche ist, wird sie durch das Graphenpaar $\mathcal{G}$ und $\mathcal{H}$ beschrieben, in denen das Produkt der längsten Wege aus beiden Graphen minimal ist.

Wimer, Koren und Cederbaum [WimerKorenCederbaum88] beschreiben einen Branch & Bound Algorithmus zur Lösung der Floorplan–Optimierung. Sie verwenden verschiedene Heuristiken, um den untersuchten Teil des Suchbaumes klein zu halten. In einem Beispiel mit

[3] Die angegebenen Werte wurden durch 21 kleinere Probleme aus einer Menge von 100 zufällig erzeugten Instanzen ermittelt, die wahren Werte beim 19–Puzzle dürften noch höher liegen.

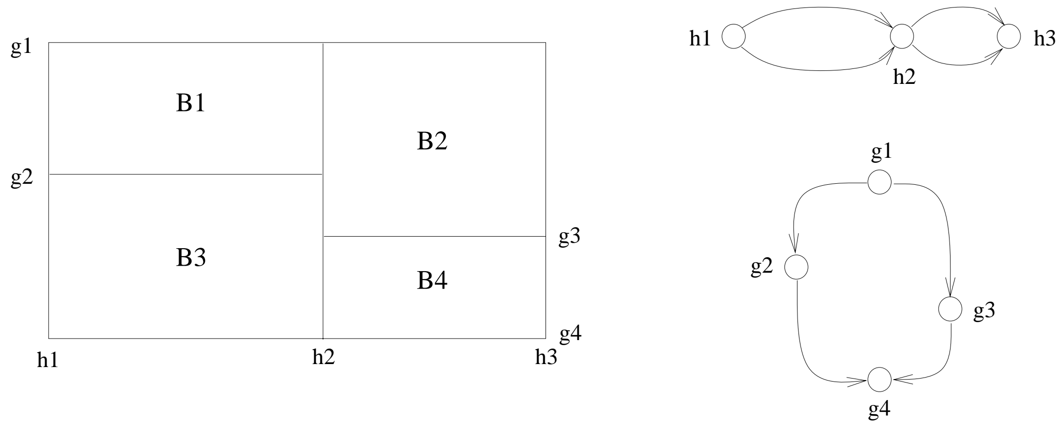


Abbildung 2.6: Ein Floorplan (links) mit den entsprechenden Graphen $\mathcal{G}$ (unten) und $\mathcal{H}$ (oben)

24 Blöcken und einer mittleren Anzahl von 4.6 Größenkombinationen je Block ergeben sich $2.03 \cdot 10^{16}$ verschiedene Konfigurationen. Ihr Branch & Bound Algorithmus betrachtet je nach verwendeter Heuristik zwischen $4.83 \cdot 10^5$ und $1.0 \cdot 10^4$ dieser Konfigurationen.

Aufgrund der großen Suchbäume und der geringen Lösungsdichte müßte der Algorithmus *IDA** sehr gut auf die Floorplan–Optimierung angewendet werden können. Die Bewertungsfunktion setzt sich dabei wie folgt zusammen:

$g(n)$ Fläche der (Teil-) Konfiguration, die durch den Knoten n beschrieben wird

$h(n)$ Fläche, um die sich der Floorplan durch das Hinzufügen der noch fehlenden Blöcke voraussichtlich vergrößert

Ein Knoten n, der in der Tiefe d im Suchbaum liegt, beschreibt eine Konfiguration, an der die Blöcke $B_1, B_2 \ldots B_d$ beteiligt sind. Die Größenkombinationen der einzelnen Blöcke werden durch den Pfad im Baum von der Wurzel bis zum Knoten n beschrieben. Abbildung 2.7 zeigt einen Floorplan und den dazugehörigen Suchbaum.

Im Gegensatz zum n×n-Puzzle ist bei der Floorplan–Optimierung der Zielzustand, d. h. die Lösungskonfiguration, noch nicht bekannt. Dagegen ist die Länge des Lösungspfades gleich der Anzahl der Blöcke im Layout und somit bekannt. Unter der Voraussetzung, daß die heuristische Funktion zulässig ist, ist somit das erste expandierte Blatt im Suchbaum ein optimaler Lösungsknoten.

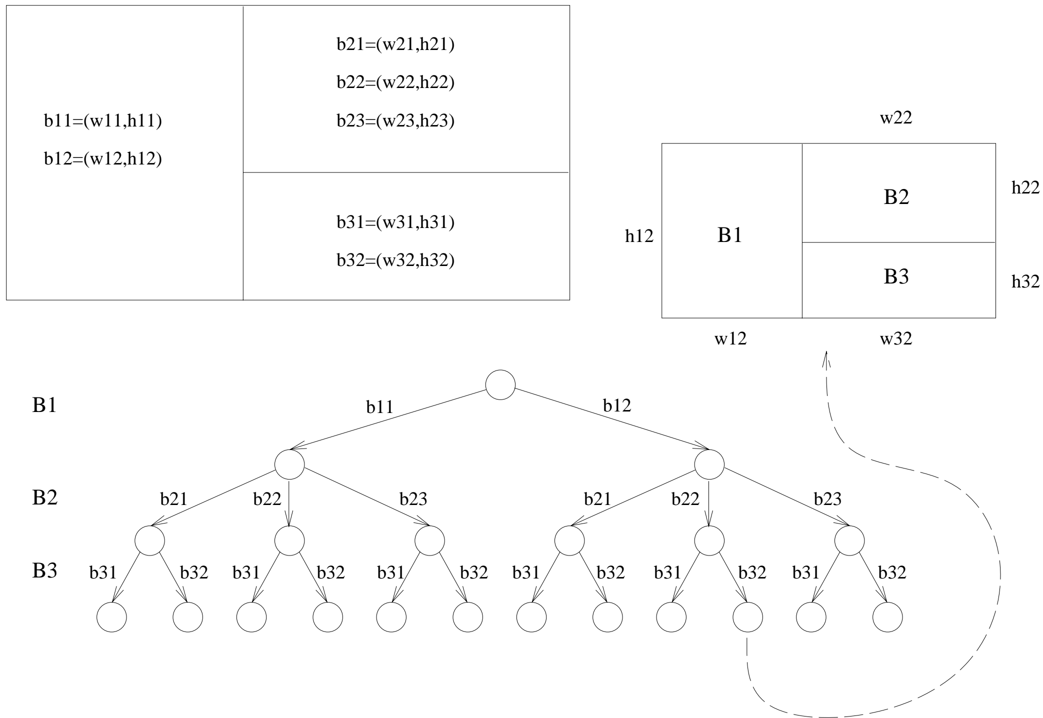


Abbildung 2.7: Ein Floorplan und der entsprechende Suchbaum

2.5.3 Das Cutting–Stock–Problem

Beim *Cutting–Stock–Problem* (Verschnitt–Problem) ist eine Grundfläche mit fester Breite gegeben, die in der Höhe unbegrenzt ist. Diese Grundfläche soll in unterschiedliche, fest vorgegebene Blöcke unterteilt werden, wobei die Höhe der benutzten Grundfläche minimal ist. Eine Abwandlung hiervon ist z. B. das zwei-dimensionale Rucksack-Problem, bei dem eine feste, nach allen Seiten begrenzte Grundfläche möglichst dicht mit bestimmten Blöcken aufgefüllt werden muß. Im Gegensatz zum Cutting–Stock–Problem sind hier jedoch mehr Blöcke vorhanden, als man auf die Grundfläche stellen kann, so daß man zusätzlich zur Anordnung der Blöcke auch noch die Auswahl der Blöcke bestimmen muß.

Das Cutting-Stock-Problem ist verwandt mit der Floorplan–Optimierung, allerdings ist die Position der Blöcke nicht vorgegeben, d. h. die Graphen, welche die Lage der Blöcke beschreiben, müssen selbst konstruiert werden, wobei die maximale Weglänge in einem der beiden Graphen begrenzt ist. Der Verzweigungsfaktor des Suchbaum ist beim Cutting–Stock–Problem größer als bei der Floorplan–Optimierung, da die Lage der Blöcke auf der Grundfläche beliebig ist.

Kapitel 3

Die Leistungsanalyse paralleler Algorithmen

Sequentielle Algorithmen können bezüglich ihrer Speicherplatz- und Zeitkomplexität analysiert und in Klassen eingeteilt werden. Diese Größen hängen von der Problemgröße ab. Bei der Leistungsanalyse paralleler Algorithmen kommt als weiterer Faktor die Zahl der verwendeten Prozessoren hinzu. Außerdem ist die Effizienz der Parallelisierung abhängig von der Architektur des Systems, auf dem die Algorithmen implementiert werden. Eine wichtige Größe, die Aufschluß über die Güte eines parallelen Algorithmus gibt, ist die *Skalierbarkeit.* Ein Parallelrechner ist skalierbar, wenn sich die Rechenleistung einfach durch ein Vergrösserung der Prozessorzahl erweitern läßt. Wie Parallelrechner sind auch parallele Algorithmen nicht beliebig skalierbar, d. h. die Rechenleistung kann nicht unbegrenzt durch eine Erhöhung der Prozessorzahl gesteigert werden. Gründe hierfür liegen in der begrenzten Parallelisierbarkeit von sequentiellen Verfahren, wie sie in dem *Gesetz von Amdahl* beschrieben wird. Hinzu kommt bei parallelen Systemen ein in der Regel mit der Prozessorzahl wachsender *Overhead*, bedingt z. B. durch die Kommunikation zwischen den Prozessoren. Mit der in diesem Kapitel vorgestellten *Isoeffizienz-Funktion* ist es möglich, die Skalierbarkeit von parallelen Systemen (Architektur und Algorithmus) zu untersuchen, und somit eine genaue Leistungsanalyse durchzuführen.

3.1 Speedup, Effizienz, Ineffizienz und Kosten

Es gibt eine Reihe von Parametern, die die Güte eines parallelen Algorithmus beschreiben. Das wohl gebräuchlichste Maß für die Güte eines parallelen Algorithmus ist der *Speedup*, der die „Beschleunigung“ des parallelen Algorithmus bei der Verwendung von p Prozessoren beschreibt.

Definition 3.1 (Speedup) *Der* **Speedup** *eines parallelen Algorithmus bei der Ausführung auf p Prozessoren berechnet sich aus dem Quotienten der Laufzeit des schnellsten sequentiellen Algorithmus* T_{seq} *durch die Laufzeit des parallelen Algorithmus* $T_{par}(p)$*, angesetzt auf das gleiche*

Problem und ermittelt auf demselben Prozessortyp:

$$sp(p) = \frac{T_{seq}}{T_{par}(p)}$$

Oftmals wird der Speedup nicht mit der Laufzeit T_{seq} des schnellsten sequentiellen Algorithmus, sondern mit der Laufzeit $T_{par}(1)$ des parallelen Algorithmus auf einem Prozessor berechnet. Diese Laufzeit ist im allgemeinen größer, da der parallele Algorithmus zusätzlichen Anweisungen für Kommunikation, Lastverteilung und Terminierungserkennung enthält. Obwohl die meisten dieser Punkte beim Lauf auf einem Prozessor unberücksichtigt bleiben, so kann der Algorithmus für diese Funktionalitäten jedoch Anweisungen beinhalten, welche auch bei einem Lauf auf einem Prozessor ausgeführt werden.

Gilt $sp(p) = p$, so spricht man von einem *optimalen Speedup*. In den meisten Fällen ist der Speedup jedoch kleiner als die Zahl der Prozessoren, da die parallele Version des Algorithmus aufwendiger ist und Kommunikation zwischen den Prozessoren hinzukommt, während der eigentliche Aufwand zur Problemlösung gleich dem der sequentiellen Version ist. Bei parallelen Baumsuchalgorithmen kann der Arbeitsaufwand jedoch stark von dem Aufwand der entsprechenden sequentiellen Algorithmen abweichen, da die Zahl der expandierten Knoten in beiden Fällen unterschiedlich ist. Bedingt durch die unterschiedliche Reihenfolge, in der die Knoten bearbeitet werden, kann die Lösung viel schneller gefunden werden, so daß sich ein überlinearer Speedup ergibt. Andererseits kann der parallele Algorithmus mehr Knoten expandieren als der entsprechende sequentielle Algorithmus. In diesem Fall ergibt sich ein sublinearer Speedup. Diese als *Speedup-Anomalien* bezeichneten Erscheinungen erschweren die Leistungsanalyse von parallelen Baumsuchalgorithmen. Ein anomalie-freier Speedup ist der *normierte Speedup*, welcher unabhängig von der Zahl der betrachteten Knoten die Leistung eines parallelen Baumsuchalgorithmus beschreibt.

Definition 3.2 (normierter Speedup) *Sei w_{seq} die Zahl der vom sequentiellen Algorithmus expandierten Knoten, $w_{par}(p)$ die Zahl der vom parallelen Algorithmus expandierten Knoten. Der* **normierte Speedup** *eines parallelen Baumsuchalgorithmus ist der Quotient aus der Zahl der vom parallelen Algorithmus in einer Zeiteinheit expandierten Knoten durch Zahl der vom sequentiellen Algorithmus in einer Zeiteinheit betrachteten Knoten:*

$$sp_{norm}(p) = \left(\frac{w_{par}(p)}{T_{par}(p)}\right) \Big/ \left(\frac{w_{seq}}{T_{seq}}\right) = \frac{w_{par}(p) \cdot T_{seq}}{T_{par}(p) \cdot w_{seq}}$$

Der Speedup sagt allerdings noch nicht sehr viel über die effektive Nutzung der Prozessoren aus, da dieser in den meisten Fällen mit der Prozessorzahl – bis zu einer bestimmten Grenze, die aber recht hoch liegen kann – wächst. Ein besseres Maß ist die *Effizienz* eines parallelen Algorithmus, welche praktisch die erste Ableitung – also die Steigung – der Speedupfunktion darstellt.

Definition 3.3 (Effizienz) *Die* **Effizienz** $eff(p)$ *eines parallelen Algorithmus ist der Quotient aus Speedup* $sp(p)$ *durch Prozessorzahl* p*:*

$$eff(p) = \frac{sp(p)}{p} = \frac{T_{seq}}{p \cdot T_{par}(p)}$$

Diese Funktion mit Wertebereich[1] $(0,1] \subset I\!R$ beschreibt die effektive Nutzung der Prozessoren, d. h. den Anteil an der Gesamtlaufzeit, in der der parallele Algorithmus dieselben Anweisungen ausführt, die auch ein sequentieller Algorithmus bei der Berechnung des gleichen Problems ausführen würde.

Kruskal, Rudolph und Snir [KruskalRudolphSnir90] definieren die *Ineffizienz-Funktion*, welches der Kehrwert der Effizienz-Funktion eines parallelen Algorithmus ist:

Definition 3.4 (Ineffizienz) *Die* **Ineffizienz** *eines parallelen Algorithmus ist der Kehrwert der Effizienz dieses Algorithmus:*

$$ineff(p) = \frac{1}{eff(p)} = \frac{p \cdot T_{par}(p)}{T_{seq}}$$

Sie teilen anhand dieser Ineffizienz-Funktion und der parallelen Laufzeit die parallelen Algorithmen in sechs Komplexitätsklassen ein und zeigen die Invarianz dieser Klassen bezüglich mehrerer Rechenmodelle.

Eine weitere Funktion zur Beschreibung der Güte eines parallelen Algorithmus führen Flatt und Kennedy [FlattKennedy89] ein:

Definition 3.5 (Kosten) *Die* **Kosten** *eines parallelen Algorithmus sind das Produkt aus Prozessorzahl* p *und paralleler Laufzeit* $T_{par}(p)$*:*

$$C(p) = p \cdot T_{par}(p)$$

Sie begründen dieses Kostenmaß damit, daß sich die Kosten für eine Berechnung immer aus der Laufzeit und den verwendeten Ressourcen zusammensetzen. Für die sequentielle Laufzeit gilt:

$$C(1) = T_{par}(1) \approx T_{seq}$$

Somit kann man die Ineffizienz als die Funktion der anteiligen Mehrkosten der Parallelisierung verstehen:

$$ineff(p) = \frac{p \cdot T_{par}(p)}{T_{seq}} = \frac{C(p)}{C(1)}$$

Je näher also die Ineffizienz an 1 liegt, desto geringer sind die „Mehrkosten“ der Parallelisierung und umso größer ist die Effizienz.

[1] Falls nicht ausdrücklich erwähnt, so wird bei den folgenden Überlegungen von einem maximal optimalen Speedup ausgegangen, d. h. die Effizienz ist niemals größer als 1. Speedup-Anomalien werden also nicht berücksichtigt.

3.2 Das Gesetz von Amdahl

Seit der Formulierung der ersten parallelen Algorithmen hat man sich Gedanken über den möglichen Speedup bei der Parallelisierung von sequentiellen Algorithmen gemacht. Eines der frühen Resultate ist die von G. M. Amdahl im Jahre 1967 formulierte und heute als *Amdahls Gesetz* bekannte Aussage, welche seitdem von Kritikern der massiv parallelen Systeme gerne zitiert wird, um die Grenzen des parallelen Rechnens aufzuzeigen:

Gesetz von Amdahl *(Amdahl's Law)*
Jedes Programm besitzt einen parallelisierbaren Anteil $\mathcal{P}$ und einen nicht parallelisierbaren Anteil $\mathcal{S}$. Es gelte $\mathcal{S} + \mathcal{P} = 1$.
Der mit p Prozessoren maximal erreichbare Speedup $sp(p)$ berechnet sich wie folgt:

$$sp(p) = \frac{T_{seq}}{T_{par}(p)} = \frac{\mathcal{S} + \mathcal{P}}{\mathcal{S} + \frac{\mathcal{P}}{p}} = \frac{1}{\mathcal{S} + \frac{\mathcal{P}}{p}}$$

Der nach Amdahl erreichbare Speedup $sp(1024)$ für einen sequentiellen Algorithmus mit einem nicht parallelisierbaren Anteil S wird in dem Graphen in Abbildung 3.2 dargestellt.

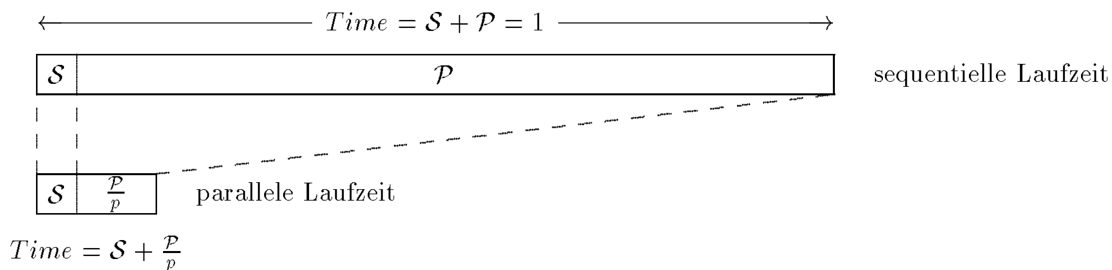


Abbildung 3.1: Modell für Speedupberechnung nach Amdahls Gesetz

Amdahls Gesetz liefert eine einfache Formel zur Berechnung einer oberen Grenze für den Speedup bei der Parallelisierung eines sequentiellen Algorithmus. Zu beachten ist jedoch, daß $\mathcal{S}$ den Anteil an der sequentiellen Laufzeit bezeichnet, der im allgemeinen nur sehr gering ist. Zum sequentiellen Anteil zählen z. B. Initialisierungen oder die Ein- und Ausgabe. Abhängig von der Architektur kann dieser Teil auf einem Parallelrechner aber auch parallelisiert werden. Der Annahme, daß der parallelisierbare Anteil $\mathcal{P}$ sich mit nur optimalem Speedup parallelisieren läßt, kann man Argumente entgegensetzen. Es verweist z. B. Annartone [Annartone92] darauf, daß in Multiprozessor-Systemen in jedem Prozessor ein Cache enthalten ist, so daß die Speicherzugriffe im parallelisierbaren Teil des Algorithmus superlinear in der Prozessorzahl schneller werden.

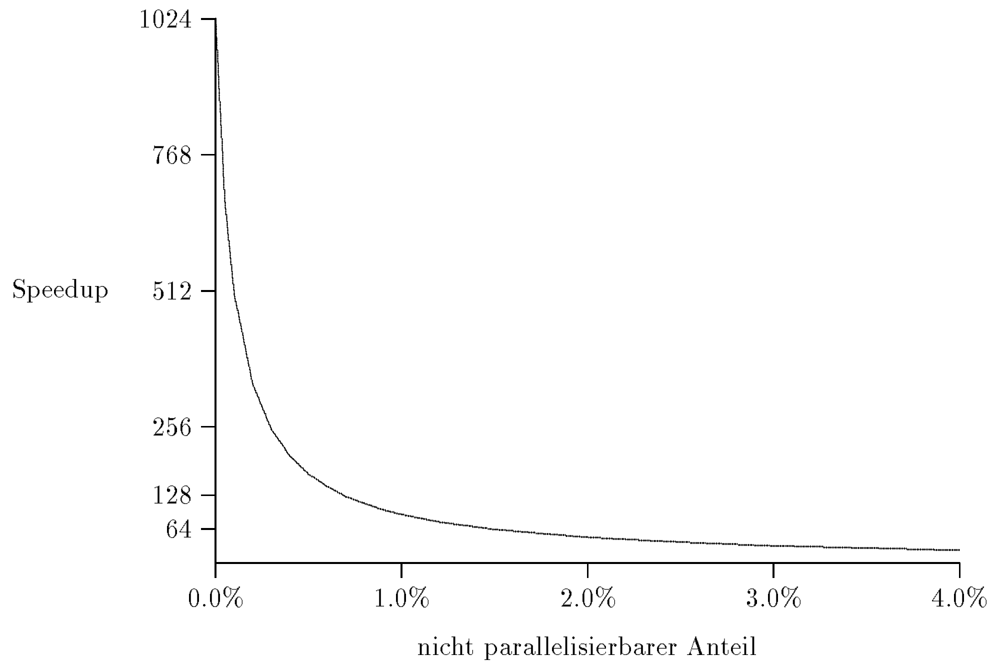


Abbildung 3.2: Der nach Amdahls Gesetz erreichbare Speedup mit 1024 Prozessoren

Gustafson [Gustafson88] berichtet über einige praktische Anwendungen, die – obwohl der sequentielle Anteil der Probleme zwischen 0.4% und 0.8% liegt – auf einem 1024-Prozessor-System Speedups zwischen 1016 und 1021 erzielten. Nach der Formel aus Amdahls Gesetz liegt die obere Schranke für den Speedup in diesen Fällen jedoch zwischen 112 und 201!

Die Betrachtungsweise von Amdahls Gesetz ist für die Informatik und auch für die Naturwissenschaft im allgemeinen untypisch: In Amdahls Gesetz wird die Problemgröße konstant gehalten und die Zahl der Prozessoren erhöht. Übertragen auf sequentielle Rechner würde dies bedeuten, daß mit den heutigen, um ein vielfaches schnelleren Rechnern, die gleichen Probleme gelöst werden wie vor zehn Jahren, bloß eben schneller! Aber dagegen können mit den heutigen Rechnern größere, schwierigere Probleme gelöst werden, deren Berechnung vor zehn Jahren noch nicht, oder nicht mit vertretbarem Aufwand möglich war.

In der Parallelverarbeitung sollte daher die Problemgröße mit der Prozessorzahl wachsen. Aus diesem Grund führt Gustafson als Maß für die Güte eines parallelen Algoritsmus den skalierten Speedup *(scaled speedup)* ein:

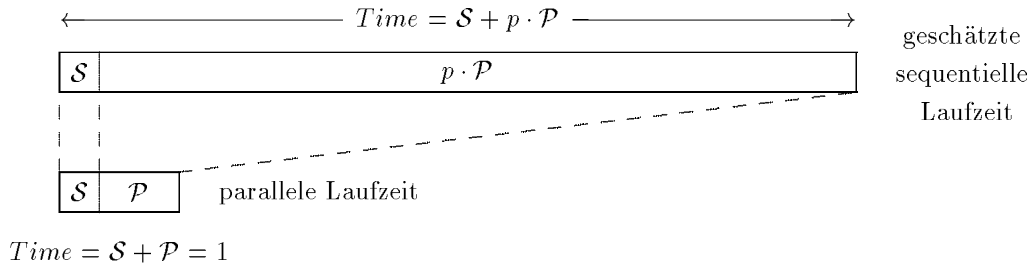


Abbildung 3.3: Modell für die Berechnung des skalierten Speedups

$$
\begin{aligned}
sp_{scaled}(p) &= \frac{\mathcal{S} + p \cdot \mathcal{P}}{\mathcal{S} + \mathcal{P}} \\
&= \mathcal{S} + p \cdot \mathcal{P} \\
&= p + (1 - p) \cdot \mathcal{S}
\end{aligned}
$$

Beim skalierten Speedup wird nicht über die sequentielle Laufzeit normiert, sondern über die parallele Laufzeit, d. h. der parallelisierbare Anteil wächst linear mit der Zahl der Prozessoren. Abbildung 3.3 im verdeutlicht Gegensatz zur Abbildung 3.1 das Modell für den skalierten Speedup.

3.3 Der Overhead eines parallelen Algorithmus

Neben dem nicht parallelisierbaren Anteil eines Algorithmus existiert eine weitere Größe, die einen optimalen Speedup im Normalfall unmöglich macht. Ein paralleles Programm enthält zusätzliche Anweisungen – etwa zur Lastverteilung und zur Terminierungserkennung –, außerdem kommen durch Kommunikation weitere Rechen- und Wartezeiten hinzu. Diese Größen kann man in dem *Overhead* eines parallelen Algorithmus zusammenfassen:

Definition 3.6 (Overhead) *Der* **Gesamtoverhead** $T_o(p)$ *eines parallelen Algorithmus bei der Ausführung auf p Prozessoren ist die Differenz zwischen Gesamtlaufzeit aller Prozessoren und der entsprechenden sequentiellen Laufzeit:*

$$T_o(p) = p \cdot T_{par}(p) - T_{seq}$$

Somit gilt für die parallele Laufzeit:

$$T_{par}(p) = \frac{T_{seq} + T_o(p)}{p}$$

Im folgenden ist $T_{par}^{comp}(p) = \frac{T_{seq}}{p}$ die *parallele Rechenzeit.* Dies ist die Ausführungszeit derjenigen Anweisungen eines parallelen Programmes, die auch das entsprechende sequentielle Programm ausführen müßte. Diese Anweisungen können nach Amdahls Gesetz jedoch nicht beliebig stark parallelisiert werden, so daß eigentlich nur gelten kann $T_{par}^{comp}(p) \approx \frac{T_{seq}}{p}$. Bei der folgenden Bestimmung der Overhead-Funktion wird jedoch von einer beliebig feinen Parallelisierbarkeit ausgegangen.

Die Overhead-Funktion kann auch in die Formeln zur Berechnung von Speedup und Effizienz aufgenommen werden:

$$
\begin{aligned}
sp(p) &= \frac{T_{seq}}{T_{par}(p)} = \frac{p \cdot T_{seq}}{T_{seq} + T_o(p)} = \frac{p}{1 + \frac{T_o(p)}{T_{seq}}} \\
eff(p) &= \frac{1}{1 + \frac{T_o(p)}{T_{seq}}}
\end{aligned}
$$

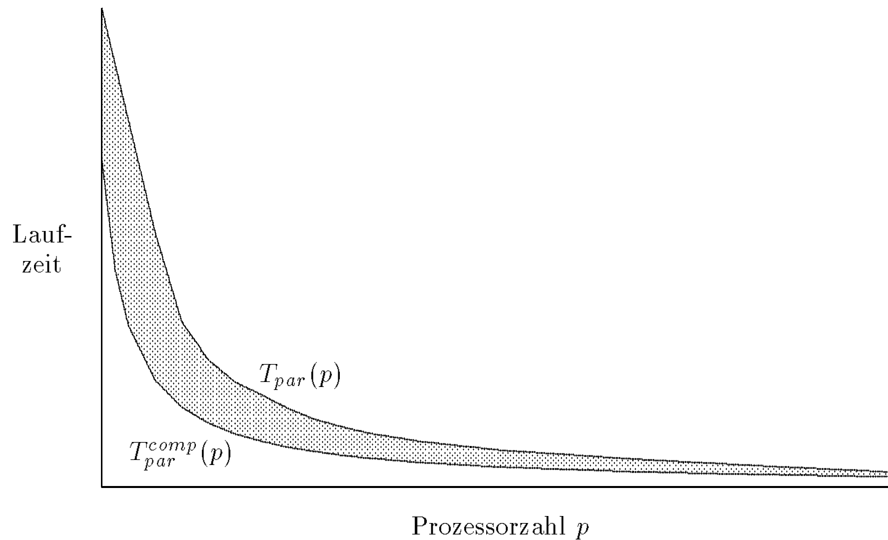


Abbildung 3.4: Der Verlauf der parallelen Laufzeit für $T_o(p) < \Theta(p)$ (der schraffierte Bereich beschreibt die Overheadzeit)

Betrachtet man nun die sequentielle Laufzeit – also die Problemgröße – als Konstante, so wird die Effizienz nur durch die Overhead-Funktion $T_o(p)$ bestimmt. Für das Wachstum dieser Funktion kann man die folgenden drei Fälle unterscheiden:

1. $\mathbf{T_o(p) < \Theta(p)}$
   Der Overhead wächst langsamer als die Prozessorzahl.

Für die parallele Laufzeit $T_{par}(p)$ gilt:

$$T_{par}(p) = \frac{T_{seq}}{p} + \frac{T_o(p)}{p} = T_{par}^{comp}(p) + \frac{T_o(p)}{p}$$

In diesem Fall ist die Overheadzeit $\frac{T_o(p)}{p}$ eine monoton fallende Funktion, die stärker fällt als die Funktion $T_{par}^{comp}(p)$, also die parallele Rechenzeit. Somit fällt auch der Anteil des Overheads an der Laufzeit mit der Prozessorzahl. Abbildung 3.4 zeigt den Verlauf der parallelen Rechenzeit (untere Kurve) und der parallelen Laufzeit (obere Kurve). Die Differenz zwischen den Werten der beiden dargestellten Kurven beschreibt die Overheadzeit $\frac{T_o(p)}{p}$.

2. $\mathbf{T_o(p) = \Theta(p)}$

   Der Overhead wächst linear mit der Prozessorzahl.
   In diesem Fall ist die Laufzeit des Overheads $\frac{T_o(p)}{p}$ konstant, da das Wachstum der Overhead-Funktion durch die erhöhte Prozessorzahl ausgeglichen wird. Da die parallele Rechenzeit $T_{par}^{comp}(p)$ nahezu beliebig klein werden kann[2], stellt der konstante Overhead $\frac{T_o(p)}{p}$ eine untere Schranke für die parallele Laufzeit dar.

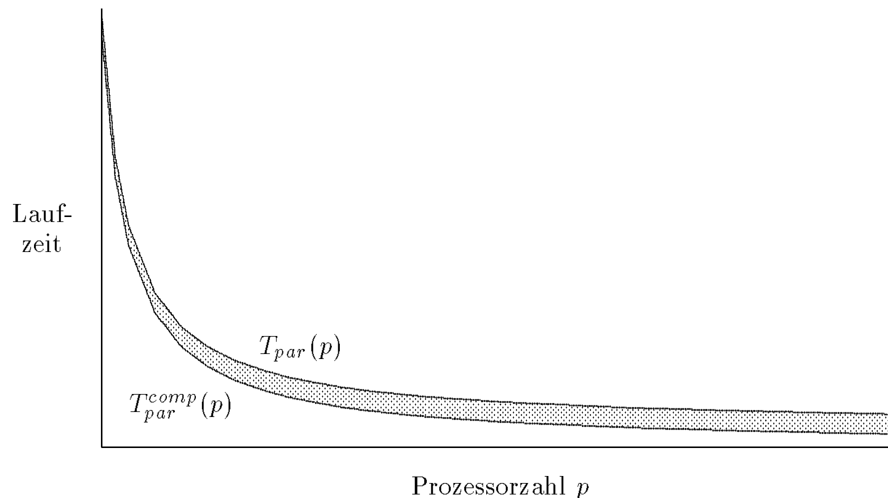


Abbildung 3.5: Der Verlauf der parallelen Laufzeit für $T_o(p) = \Theta(p)$

3. $\mathbf{T_o(p) > \Theta(p)}$

   Die Funktion $T_o(p)$ wächst überlinear in der Prozessorzahl.
   In diesem Fall wächst die Overheadzeit $\frac{T_o(p)}{p}$ mit steigender Prozessorzahl. Die parallele Laufzeit $T_{par}(p)$ fällt bis zu einer minimalen Laufzeit T_{par}^{min}. Danach steigt die Laufzeit des parallelen Algorithmus für wachsende Prozessorzahlen (vgl. Abbildung 3.6).

[2]Amdahls Gesetz bestimmt hier eine untere Schranke.

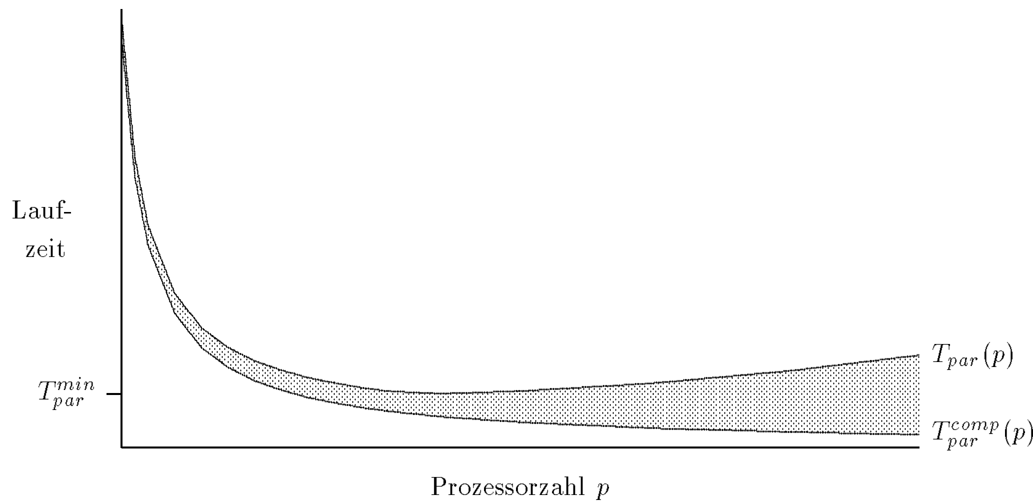


Abbildung 3.6: Der Verlauf der parallelen Laufzeit für $T_o(p) = \Theta(p)$

3.4 Die Isoeffizienz-Funktion

Bei der Leistungsanalyse eines parallelen Algorithmus kann die Architektur des Rechners, auf dem der Algorithmus implementiert wird, nicht unberücksichtigt bleiben. Die Architektur bestimmt maßgeblich die Overhead-Funktion des Algorithmus.

Definition 3.7 (paralleles System) *Ein* **paralleles System** *bezeichnet die Kombination aus einem parallelen Algorithmus mit einer bestimmten parallelen Architektur.*

Die Effizienz eines parallelen Algorithmus sinkt mit steigender Prozessorzahl bei konstanter Problemgröße. Gründe hierfür liegen in der Overhead-Funktion und in der durch Amdahls Gesetz beschriebenen begrenzten Parallelisierbarkeit von Problemen.

Für die Effizienz $eff(p)$ gilt die folgende im Kapitel 3.3 berechnete Funktion:

$$eff(p) = \frac{1}{1 + \frac{T_o(p)}{T_{seq}}}$$

Um nun eine konstante Effizienz E für ein paralleles System zu gewährleisten, muß das Wachstum der Overhead-Funktion für größere Prozessorzahlen p durch Vergrößern der Problemgröße $W(p)$ – welche proportional zur sequentiellen Laufzeit T_{seq} ist – aufgefangen werden.

Es gilt somit:

$$\begin{aligned} E &= \frac{1}{1+\frac{T_o(p)}{T_{seq}(p)}} \\ \Longleftrightarrow \quad 1+\frac{T_o(p)}{T_{seq}} &= \frac{1}{E} \\ \Longleftrightarrow \quad \frac{T_o(p)}{T_{seq}} &= \frac{1-E}{E} \\ \Longleftrightarrow \quad T_{seq} &= \frac{E}{1-E}\cdot T_o(p) \\ \Longrightarrow \quad & \boxed{W(p) = \Omega\left(\frac{E}{1-E}\cdot T_o(p)\right)} \end{aligned}$$

Der Ausdruck $\frac{E}{1-E}\cdot T_o(p)$ beschreibt das Wachstum der Problemgröße, welches nötig ist, um eine konstante Effizienz E für eine beliebige Prozessorzahl p eines parallelen Systems zu halten. Diese von Kumar und Rao [KumarRao90] eingeführte Funktion wird als *Isoeffizienz-Funktion* eines parallelen Systems bezeichnet.

Definition 3.8 (Isoeffizienz-Funktion) *Sei $W(p)$ die Problemgröße beim Lauf auf p Prozessoren, E eine konstante angestrebte Effizienz. Die* **Isoeffizienz-Funktion** *$f(p)$ beschreibt das Wachstum der Problemgröße, welches nötig ist, um die geforderte Effizienz E für beliebige Prozessorzahlen p zu gewährleisten:*

$$W(p) = \Omega(f(p)) \Longrightarrow eff(p) \geq E$$

Diese Funktion beschreibt die Skalierbarkeit eines parallelen Systems. Eine gering wachsende Isoeffizienz-Funktion deutet auf eine gute Skalierbarkeit hin, da die Problemgröße nicht sehr stark – im Idealfall nur linear – in der Prozessorzahl wachsen muß, um eine konstante Effizienz zu gewährleisten. Eine stark – etwa exponentiell – wachsende Isoeffizienz-Funktion deutet auf eine schlechte Skalierbarkeit des Systems hin, da in diesem Fall die Problemgröße exponentiell in der Prozessorzahl wachsen muß, damit die Effizienz konstant bleibt.

Eine untere Schranke für die Isoeffizienz-Funktion ist $\Omega(p)$. Eine geringer wachsende Isoeffizienz-Funktion würde bedeuten, daß ab einer bestimmten Prozessorzahl die Parallelisierbarkeit des Problems erschöpft ist, da im System mehr Prozessoren existieren als unteilbare Arbeitsanteile $W(p)$. Hieraus folgt, daß $\Theta(p)$ die Isoeffizienz-Funktion eines ideal-skalierbaren parallelen Systems ist.

Die im Kapitel 3.2 beschriebenen parallelisierbaren und nicht parallelisierbaren Anteile $\mathcal{P}$ und $\mathcal{S}$ können zusätzlich zur Overhead-Funktion für die Berechnung der Isoeffizienz-Funktion

herangezogen werden. Für die parallele Laufzeit mit p Prozessoren gilt nun:

$$\begin{aligned} T_{par}(p) &= \frac{T_{seq} \cdot (\mathcal{S} + \mathcal{P})}{p} + \frac{T_o(p)}{p} \\ &= T_{seq} \cdot \mathcal{S} + \frac{\mathcal{P} \cdot T_{seq}}{p} + \frac{T_o(p)}{p} \end{aligned}$$

Für die Berechnung der Isoeffizienz-Funktion muß festgelegt werden, wie der sequentielle und der parallelisierbare Anteil mit der Problemgröße wachsen. Hier wird davon ausgegangen, daß die absolute Laufzeit T_s des sequentiellen Teils konstant bleibt, d. h. mit der Problemgröße wächst nur der parallelisierbare Anteil des Problems. Dieses erscheint sinnvoll, da der in Amdahls Gesetz beschriebene nicht parallelisierbare Anteil gerade die Ein-/Ausgabe bzw. Initialisierungen beinhaltet. Diese sind abhängig von dem betrachteten Problem bzw. Algorithmus, nicht jedoch abhängig von der Problemgröße. Das hat zur Folge, daß der relative sequentielle Anteil $\mathcal{S}$ für größere Probleme kleiner wird.

Es gilt:

$$T_{seq} = T_{seq} \cdot \mathcal{P} + T_{seq} \cdot \mathcal{S} = T_{seq} \cdot \mathcal{P} + T_s$$

$$\Longrightarrow \qquad \mathcal{P} = 1 - \frac{T_s}{T_{seq}} \quad \text{und} \quad \mathcal{S} = \frac{T_s}{T_{seq}}$$

Eingesetzt in die Formel für die Effizienz ein, erhält man die Isoeffizienz-Funktion für eine konstante Effizienz E betrachtet:

$$\begin{aligned} E &= \frac{1}{p \cdot \mathcal{S} + \mathcal{P} + \frac{T_o(p)}{T_{seq}}} \\ \Longleftrightarrow \quad E &= \frac{1}{p \cdot \frac{T_s}{T_{seq}} + 1 - \frac{T_s}{T_{seq}} + \frac{T_o(p)}{T_{seq}}} \\ \Longleftrightarrow \quad E &= \frac{T_{seq}}{p \cdot T_s + T_{seq} - T_s + T_o(p)} \\ \Longleftrightarrow \quad T_{seq} &= E \cdot ((p-1) \cdot T_s + T_{seq} + T_o(p)) \\ \Longleftrightarrow \quad T_{seq} &= \frac{E}{1-E} \cdot ((p-1) \cdot T_s + T_o(p)) \end{aligned}$$

Die Funktion

$$\boxed{W(p) = \Omega\left(\frac{E}{1-E} \cdot ((p-1) \cdot T_s + T_o(p))\right)}$$

liefert eine allgemeine Formel zur Berechnung der Isoeffizienz-Funktion eines parallelen Systems, wobei E die zu erreichende Effizienz, T_s die absolute (konstante) Laufzeit des sequentiellen Anteils und $T_o(p)$ die Overheadfunktion bezeichnen.

Kapitel 4

Parallelisierungsansätze der iterativen Tiefensuche

Die iterative Tiefensuche eignet sich sehr gut zur Parallelisierung, da das Verfahren eine sehr feine Parallelität enthält und die Suchbäume sehr groß sind. Man kann einen Suchbaum beliebig in disjunkte Teilbäume zerlegen, welche dann unabhängig voneinander von den einzelnen Prozessoren durchsucht werden. Aufgrund der Ungleichmäßigkeit der Suchbäume ist die *Lastverteilung* entscheidend für die Effizienz eines parallelen Baumsuchverfahrens. Hierbei müssen zwei Architekturklassen (*MIMD* und *SIMD*) unterschieden werden. Für die MIMD-Systeme stellten Kumar und Rao im Jahre 1987 in ihrem Algorithmus *PIDA** einen Parallelisierungsansatz vor, welchen sie später auch bei der Parallelisierung von Branch&Bound-Verfahren verwendeten. Für SIMD-Systeme entwickelten Korf, Powley und Ferguson den Algorithmus *SIDA**. In diesem Kapitel werden nach einer kurzen Beschreibung der Lastverteilung die wichtigsten der bisherigen Ansätze zur Parallelisierung der iterativen Tiefensuche vorgestellt.

4.1 Die Lastverteilung in parallelen Baumsuchverfahren

Das Hauptproblem bei der Parallelisierung von Baumsuchverfahren liegt in der *Lastverteilung*. Aufgabe der Lastverteilung ist das Aufteilen des Suchbaumes unter den Prozessoren, wobei unterschieden werden kann zwischen der *Initialverteilung* und der *dynamischen Lastverteilung* während der Suche. Aufgabe der Initialverteilung ist es, zu Beginn der Suche alle Prozessoren im System möglichst schnell mit Arbeit zu versorgen. Ideal wäre eine schon in dieser Phase gleichmäßige Aufteilung des Suchbaumes unter den Prozessoren. Solch eine Verteilung ist jedoch nicht möglich, da die Suchbäume sehr unregelmäßig sind und die Größen der bei einer Aufteilung entstandenen Teilbäume nicht *a priori* bekannt sind. Da die einzelnen Prozessoren somit unterschiedlich große Teilbäume in der Initialverteilung erhalten, ist während der weiteren Suche eine dynamische Lastverteilung nötig. In der dynamischen Lastverteilungsphase müssen diejenigen Prozessoren, die eine Überlast (also einen großen Teilbaum) haben, Arbeit

an diejenigen Prozessoren abgeben, welche ihren zugewiesenen Teilbaum schon durchsucht haben. Diese Lastverteilung trägt natürlich zum Overhead eines parallelen Algorithmus bei und beeinträchtigt dessen Leistung.

Man kann die Parallelrechner in Klassen einteilen, wobei die Klassen *MIMD* und *SIMD* die meisten der heutigen Parallelrechner enthalten. In MIMD- *(Multiple Instruction Multiple Data)* Systemen arbeiten die Prozessoren unabhängig voneinander und kommunizieren über einen gemeinsamen Speicher oder über feste Kanäle miteinander, wobei jeder Prozessor einen lokalen Speicher hat. In SIMD- *(Single Instruction Multiple Data)* Systemen arbeiten alle Prozessoren synchron dasselbe Programm ab. Im Gegensatz zu den MIMD-Systemen müssen somit alle Prozessoren dieselbe Anweisung ausführen. Dieses erschwert die dynamische Lastverteilung. Ist die Last neu zu verteilen, so müssen alle Prozessoren die Suche unterbrechen, während bei MIMD-Systemen nur diejenigen Prozessoren an der Lastverteilung beteiligt sind, die Arbeit abgeben bzw. Anfragen nach Arbeit bearbeiten.

4.2 MIMD–Rechner

4.2.1 PIDA* – Parallel IDA*

Einer der ersten Ansätze, die iterative Tiefensuche zu parallelisieren, stammt von Kumar, Rao und Ramesh aus dem Jahre 1987 [RaoKumarRamesh87, KumarRao90]. Der von ihnen vorgestellte Algorithmus *PIDA** arbeitet mit *Baumpartitionierung* und dynamischer Lastverteilung mittels *Auftragsanziehung* und diente seitdem mehreren Autoren (z. B. [Reinefeld92], [FarrageMarsland93]) als Vorlage für ihre Parallelisierungen und liefert für kleine Netze gute Effizienzen.

In ihrem Algorithmus *PIDA** erhält zu Beginn nur ein Prozessor P_0 die Wurzel des Suchbaumes zugewiesen. Dieser Prozessor beginnt dann damit, den Suchbaum zu entwickeln und die dabei noch nicht bearbeiteten Brüder der untersuchten Knoten auf einem Stack abzulegen. Alle anderen Prozessoren forden durch eine Anfrage an einen ihrer Nachbarn Arbeit, d. h. Knoten des Suchbaumes, an *(Auftragsanziehung)*. Wenn der Stack des Prozessors P_0 eine bestimmte Größe erreicht hat, wird ein Teil davon an einen Nachbarprozessor P_1 abgegeben, der nun auch mit der Suche beginnen kann. Dieses Verfahren setzt sich durch das gesamte Netz fort, bis jeder Prozessor Knoten des Suchbaumes zur weiteren Expansion erhalten hat.

Hat ein Prozessor seinen gesamten Stack abgearbeitet, so stellt er wiederum eine Anfrage an einen Nachbarprozessor. Falls dieser genügend unbearbeitete Knoten auf seinem Stack hält, läßt er ihm durch Transfer einiger Knoten, d. h. durch Transfer eines Teils seines Stacks, neue Arbeit zukommen. Hat der Nachbarprozessor keine unbearbeiteten Knoten auf seinem Stack, so teilt er dieses durch eine Rückantwort dem anfragenden Prozessor mit. Dieser schickt dann

eine weitere Anfrage an einen anderen Nachbarprozessor, wobei alle Nachbarn eines Prozessors in einem Round–Robin Verfahren bei der Suche nach neuer Arbeit berücksichtigt werden. Falls kein Nachbarprozessor Arbeit abgeben konnte, so wird eine auf dem Verfahren von Dijkstra [DijkstraFeijenGastern83] basierende Terminierungserkennung durchgeführt. Diese Terminierungserkennung findet am Ende jeder Iteration statt, so daß hier alle Prozessoren des Systems synchronisiert werden.

Die Isoeffizienz-Funktion für dieses Verfahren wird maßgeblich durch die Kommunikationszeit für die Arbeitsverteilung bestimmt. Der Anteil dieser Kommunikationszeit an der Gesamtlaufzeit ist für Netze mit kleinem Grad[1], wie z. B. dem Ring sehr groß. Dieses führt dazu, daß eine initiale, einigermaßen gleichmäßige Verteilung der Arbeit auf die Prozessoren sehr kommunikations- und zeitintensiv ist, so daß dieses Verfahren auf großen Netzen nur sehr schlechte Speedups liefert(genaue Berechnung der Isoeffizienz-Funktion für *PIDA** in Kapitel 7.1.1).

Kumar und Rao implementierten ihren Algorithmus *PIDA** auf einem 128 Prozessor Intel iPSC Hypercube, einem 120 Prozessor BBN Butterfly und einem 30 Prozessor Sequent Balance. Sie testeten ihre Implementierung mit einigen ausgewählten Instanzen der 100 Stellungen des 15-Puzzles aus [Korf85]. Sie erreichten einen Speedup von 63.5 für einen Ring[2] mit 128 Prozessoren (realisiert durch eine Einbettung in den Intel Hypercube) und einen Speedup von 115 auf dem Hypercube mit 128 Prozessoren. Zur Vermeidung von Speedup-Anomalien (siehe Kapitel 3.1) wurden alle Lösungen des jeweiligen Problems gesucht, so daß die Zahl der von IDA* und der parallelen Variante expandierten Knoten identisch ist.

4.2.2 Ähnliche Ansätze

Bauer und Krieter [BauerKrieter88] beschreiben eine Parallelisierung der iterativen Tiefensuche, in der ein Arbeitsanziehungsverfahren ähnlich wie im Algorithmus *PIDA** verwendet wird. Dabei werden neben den direkten Nachbarn auch Anfragen an Prozessoren gestellt, die im Netz einen Abstand von zwei haben. Bei der Initialverteilung wird außerdem eine statische Verteilung über einen zuvor festgelegten Prozessorbaum implementiert. Für die Terminierungserkennung zwischen den Iterationen werden sowohl ein Markierungsverfahren nach [DijkstraFeijenGastern83] als auch ein Verfahren getestet, in dem die Prozessoren nach erfolglosen Anfragen selbstständig die Arbeit einstellen und dieses einem Master-Prozessor mitteilen. Ihre Algorithmen wurden in *OCCAM* auf einem Transputer-System mit 32 Prozessoren imple-

[1] Der *Grad* eines Netzes ist die maximale Anzahl der Nachbarn eines Prozessors im Netz.

[2] Werden beim Ring nur an die direkten Nachbarn Knoten weitergegeben, so erzielten Kumar und Rao nur einen Speedup von 25 auf 128 Prozessoren. Ein Speedup von 63.5 konnte nur erreicht werden, wenn Knoten an beliebige Prozessoren im Ring weitergegeben werden.

mentiert, wobei die Prozessoren in einer Ring-Topologie mit einem zusätzlichen unregelmäßigen Chordal-Ring verschaltet wurden. Sie erzielten für 15 Instanzen des 15-Puzzles im Durchschnitt einen anomalie-freien Speedup von über 30, was einer Effizienz von 95% entspricht.

Farrage und Marsland [FarrageMarsland93] implementierten das Verfahren von Kumar und Rao auf einem Workstation-Cluster. Sie untersuchen unterschiedliche Stack-Aufteilungsstrategien und analysieren deren Auswirkungen auf den Kommunikationsanteil. Die besten Ergebnisse erzielen sie, wenn zur Lastverteilung Knoten aus dem oberen Viertel des Suchbaumes abgegeben werden. Sie führten Untersuchungen anhand von vier Instanzen des 15-Puzzles durch und berichten von einem Speedup von 15.6 auf einem Netz mit 16 Workstations.

4.2.3 PWS – Parallel Window Search

Korf, Powley und Ferguson [KorfPowleyFerguson90] beschreiben einen anderen Parallelisierungsansatz. Ihre *parallele Fenstersuche* stammt aus dem Bereich der Spielbaumsuche. In diesem Verfahren durchsuchen alle Prozessoren parallel den gesamten Suchbaum, jedoch mit unterschiedlichen Kostenschranken (Fenstern). Im Gegensatz zu den obigen Parallelisierungsansätzen kann bei diesem Verfahren jedoch nicht die Optimalität einer gefundenen Lösung garantiert werden. Der Vorteil bei diesem Verfahren ist jedoch, daß relativ schnell ein Lösungsknoten gefunden wird. Um die Optimalität einer Lösung zu garantieren, muß jedoch die Iteration mit der nächst-kleineren Suchschwelle oder – falls diese nicht ermittelt werden kann[3] – mit dem Wert $f(n)$ des gefundenen Lösungsknotens als Kostenschranke komplett durchgeführt werden. Da die Probleme, die mit dem Algorithmus IDA* gelöst werden, jedoch in der Regel eine geringe Lösungsdichte haben, ist die parallele Fenstersuche zur Implementierung einer parallelen Version der heuristischen iterativen Tiefensuche ungeeignet. Diejenigen Prozessoren, die an Iterationen mit einer höheren Kostenschranke als der Lösung arbeiteten, durchlaufen Bereiche des Suchbaumes, die im sequentiellen Fall nicht durchsucht werden.

4.3 SIMD–Rechner

4.3.1 SIDA* – IDA* für SIMD–Architekturen

Powley, Ferguson und Korf beschreiben in [PowleyFergusonKorf93] eine parallele Version von *IDA** für SIMD-Rechner. Ihr Algorithmus *SIDA** wurde auf einer *Connection-Machine CM-2* mit 32768 Prozessoren implementiert.

Der Algorithmus *SIDA** besteht aus zwei Phasen:

[3] Beim Puzzleproblem ist das Inkrement der Suchschwelle zwischen zwei Iterationen genau 2, in der allgemeinen Implementierung des Algorithmus *IDA** kann dieser Wert jedoch nicht *a priori* bestimmt werden.

1. Zu Beginn hält nur ein Prozessor die Wurzel des Suchbaumes. Er erzeugt deren Nachfolger und gibt diese – bis auf einen Knoten – an andere Prozessoren weiter. Dieses Verfahren setzt sich fort, bis alle Prozessoren einen Ausgangsknoten für ihre Suche erhalten haben. Dieses als *Breadth-First-Allocation* bezeichnete Verfahren wurde von den Autoren schon in [KorfPowleyFerguson90] vorgestellt und analysiert.

2. In der zweiten Phase führt jeder Prozessor die iterative Tiefensuche durch. Der gesamte Suchbaum wird durchsucht, indem jeder Prozessor seinen eigenen Teilbaum expandiert, ausgehend von dem Knoten, den er in der ersten Phase erhalten hat.

Die einzelnen Iterationen mit Tiefensuche bis zur Kostenschranke werden synchron durchgeführt, d. h. ein Prozesor kann nach der Abarbeitung seines Teilbaumes nicht mit der nächsten Iteration beginnen, sondern muß warten, bis alle Prozessoren ihre Suchbäume bis zur aktuellen Tiefe bearbeitet haben. Die Größe der Teilbäume ist sehr unterschiedlich, so daß eine effiziente Lastverteilungsstrategie für das Verfahren gefunden werden muß. Da der Algorithmus auf einer SIMD-Maschine implementiert wurde, kommt als weiteres Problem hinzu, daß zu einem Zeitpunkt alle Prozessoren entweder ihre Teilbäume durchsuchen oder die Last umverteilen. Hierzu wird periodisch der Anteil der Prozessoren ermittelt, die ihre Teilbäume komplett abgearbeitet haben. Übersteigt dieser Anteil einen bestimmten Schwellwert, so muß die Suche durch eine Lastverteilungsphase unterbrochen werden, wobei die größte Effizienz mit einem dynamischen Schwellwertes erreicht wird.

Im Algorithmus *SIDA** werden Informationen aus der vorherigen Iteration benutzt, um die initiale Verteilung duch Knotenexpansion und Knotenkontraktion[4] zu verbessern. Die sequentielle Laufzeit, welche zur Speedupberechnung benutzt wird, muß im Fall von *SIDA** berechnet werden, da die niedrige Zahl von 322 Knotenexpansionen pro Sekunde mit einem CM-2 Prozessor (T805: 35000 Knotenexpansionen pro Sekunde) eine sequentielle Rechenzeit für die 100 Probleme aus [Korf85] 31377 Stunden bzw. 3.6 Jahren bedeutet!

In den empirischen Untersuchungen erzielten sie die folgenden Ergebnisse für die Lösung aller 100 Instanzen des 15-Puzzles aus [Korf85]:

p	$sp(p)$	$eff(p)$	$T_{par}(p)$
8 k	5685	69%	–
16 k	10435	64%	–
32 k	17300	53%	8081 sec = 134.7 min

Als Isoeffizienz-Funktion für SIDA* auf der Connection-Machine geben sie $\mathcal{O}(p \cdot \log p)$ an,

[4]Die *Knotenkontraktion* ist die Gegenoperation zur Knotenexpansion. Bei der Kontraktion werden alle Nachfolger eines Knotens wieder zu ihrem Vorgänger vereinigt.

was auf eine gute Skalierbarkeit des Algorithmus hindeutet.

In der Zusammenfassung schreiben Powley, Korf und Ferguson: "*... these are the largest reported speedups for this algorithm and domain, and together with the high degree of scalability, demonstrate that SIMD-machines can be effectively used to traverse irregular, dynamically generated trees.*" Die Autoren haben zwar gezeigt, daß es möglich ist, auch auf einem SIMD-Rechner einen parallelen Baumsuchalgorithmus zu implementieren, aber aufgrund des unregelmäßigen Aufbaus der Suchbäume ist der Anteil der Lastverteilung an der Gesamtlaufzeit ungemein hoch, da immer alle Prozessoren an der Lastverteilung beteiligt sind, bzw. die Suche unterbrechen müssen. In asynchronen Systemen kann dagegen ein Prozessor, der seinen Teilbaum abgearbeitet hat, sofort neue Arbeit von anderen Prozessoren anfordern. Dabei müssen nur derjenige Prozessor, der die Arbeit abgibt und eventuell die Prozessoren, die an der nötigen Kommunikation beteiligt sind, ihre Suche unterbrechen. Dieses zeigt sich in den im Falle der MIMD-Systeme erreichten besseren Laufzeiten im Gegensatz zum Algorithmus *SIDA** auf einem SIMD-System. Außerdem hat die sehr geringe Zahl von 322 Knotenexpansionen pro Sekunde auf einem Connection-Machine Prozessor auch bei 32768 Prozessoren und einer Effizienz von 53% immer noch eine Laufzeit von 134 Minuten für die 100 Probleme aus [Korf85] zur Folge. Der in Kapitel 5 vorgestellte Algorithmus *AIDA** benötigt bei einer Effizienz von nur 68.7% mit 1024 Prozessoren auf dem MIMD-Rechner Parsytec GCel für die gleichen Probleme nur etwa 24 Minuten.

Kapitel 5

Der Algorithmus AIDA*

Der in diesem Kapitel vorgestellte Algorithmus *AIDA** ist ein iterativer Tiefensuchalgorithmus für asynchrone Parallelrechner. Der Algorithmus nutzt die Vorteile der MIMD-Architektur und der iterativen Tiefensuche und hat einen geringen Kommunikationsoverhead. Die wichtigsten Eigenschaften dieses Algorithmus sind:

- Die Initialverteilung erfordert – bis auf die Verbreitung der Wurzel des Suchbaumes im System – keinerlei Kommunikation oder Synchronisation.
- Die Prozessoren im System arbeiten asynchron an unterschiedlichen Iterationen, es gibt keine globale Synchronisation zwischen den Iterationen.
- Es existiert eine feste Aufteilung des Suchbaumes in disjunkte Teilbäume, welche über alle Iterationen bestehen bleibt. Dadurch können Informationen über jeden Teilbaum gesammelt werden, welche in den weiteren Iterationen von Nutzen sind.

Neben der Beschreibung des Algorithmus wird am Ende des Kapitels auf Verbesserungsansätze eingegangen.

5.1 Die Architektur des Algorithmus AIDA*

Der Algorithmus *AIDA** *(Asynchronous Parallel Iterative Deepening A*)* besteht aus drei Phasen:

1. In der Initialphase expandieren alle Prozessoren redundant den gleichen Suchbaum bis zu einer bestimmten Tiefe und erhalten eine Suchfront, die als Grundlage einer guten Initialverteilung dient.
2. In der zweiten Phase findet eine statische Abbildung der Suchfront auf die Prozessoren statt. Danach expandiert jeder Prozessor seine Knoten bis zu einer bestimmten Schwelle und speichert wiederum die sich dabei ergebende Suchfront.

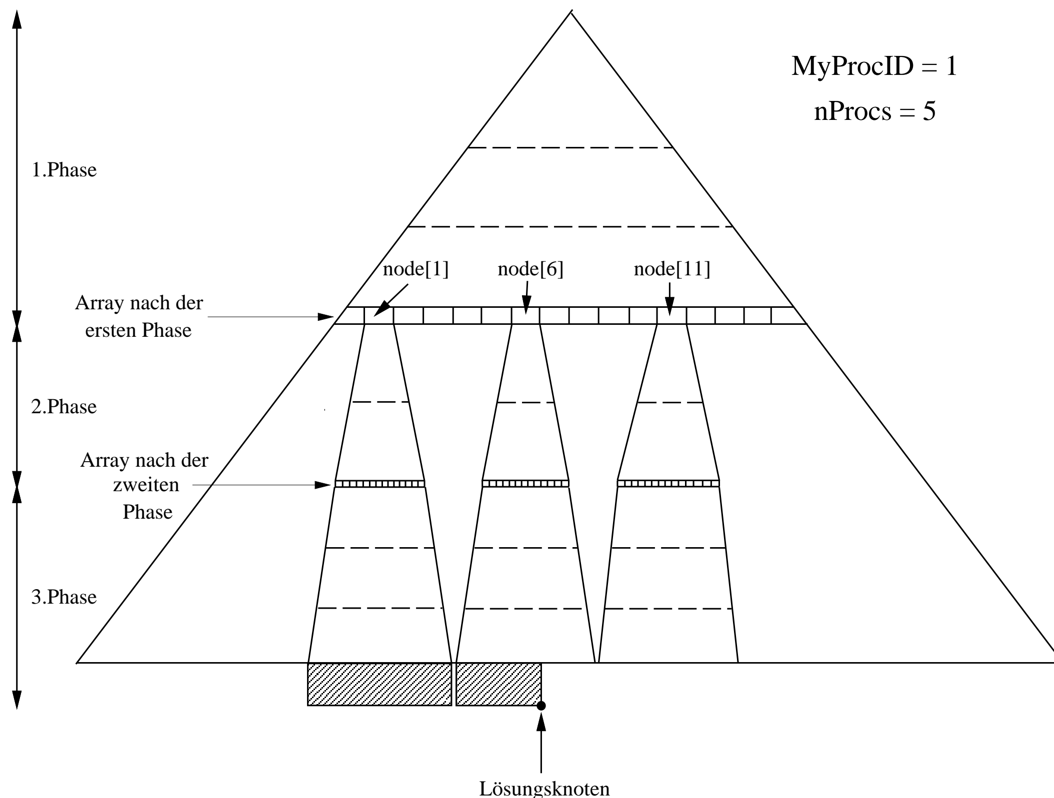


Abbildung 5.1: Die drei Phasen des Algorithmus AIDA*

3. In der eigentlichen Suchphase expandiert jeder Prozessor die in seinem lokalen Speicher gehaltenen Knoten. Diese Expansion erfolgt durch Anwendung der sequentiellen Tiefensuchroutine mit den gespeicherten Knoten als Wurzeln der zu untersuchenden Teilbäume. Sobald ein Prozessor alle Knoten im Array abgearbeitet hat, erfolgt am Ende jeder Iteration eine Lastverteilungsphase. Falls im System keine unbearbeiteten Knoten für diese Iteration vorhanden sind, beginnen alle Prozessoren unabhängig voneinander mit der nächsten Iteration, bis eine Lösung gefunden wird.

Da in der dritten Phase der Reihe nach disjunkte Teilbäume mit den von einem Prozessor lokal gehaltenen Knoten als Wurzeln durchsucht werden, kann dieses mit der Tiefensuchroutine des sequentiellen *IDA**-Algorithmus durchgeführt werden. Es entsteht daher – im Gegensatz zu den Parallelisierungen von Bauer und Krieter [BauerKrieter88], Kumar und Rao [KumarRao90], Reinefeld [Reinefeld92] und Farrage und Marsland [FarrageMarsland93] – in der Routine zur Knotenexpansion kein zusätzlicher Verwaltungsaufwand. Somit ist die Knotenexpansionszeit in *AIDA** gleich der Expansionszeit in einer entsprechenden Implementierung des sequentiellen Algorithmus *IDA**. Dieses ist ein wesentlicher Unterschied zu den im Kapitel 4 beschriebenen

Parallelisierungsansätzen. Durch die Verwendung der sequentiellen Expansionsroutine lassen sich andere Implementierungen der iterative Tiefensuchen ohne großen Aufwand auf *AIDA** übertragen und somit einfach parallelisieren.

5.2 Die erste Phase: Synchrone Expansion

```
IDA_store ( node, g )
 for all successors succ of node do
   if f (node.succ) ≤ threshold
      then if not solved
           then IDA_store ( node.succ, g(node.succ) );
           else output solution_path and exit;
      else store ( node.succ, g(node.succ) ) in local array;

first_phase ( root )
 threshold := h ( root );
 while number_stored_nodes < MAX_NODES do
   IDA_store ( root, 0 );
   increment ( threshold );
```

Abbildung 5.2: Die erste Phase von *AIDA**

Zu Beginn der ersten Phase von *AIDA** wird die Wurzel des Suchbaumes im System verbreitet. Danach beginnt jeder Prozessor, den Suchbaum durch Ausführung der rekursiven Routine `IDA_store` (vgl. Abbildung 5.2) bis zu einer bestimmten Schwelle zu entwickeln, wobei die Suchfront in einem lokalen Array gespeichert wird. Die Größe des in der ersten Phase entwickelten Teilbaumes bzw. die Anzahl der im lokalen Array eines jeden Prozessors gespeicherten Knoten wird durch die Wahl der Konstanten `MAX_NODES` bestimmt. Am Ende der ersten Phase befinden sich in den lokalen Arrays aller Prozessoren exakt dieselben Knoten.

5.3 Die zweite Phase: Statische Lastverteilung und parallele Expansion

Zu Beginn der zweiten Phase findet eine Abbildung aller Knoten der gespeicherten Suchfront auf die Prozessoren des Systems statt, indem jeder Prozessor seinen Anteil aus der Suchfront auswählt. Die dabei erhaltenen Knoten werden in dieser Phase als Wurzeln der Suchbäume aufgefaßt, die mittels iterativer Tiefensuche von einem Prozessor durchsucht werden. Nach der Aufteilung der Suchfront expandiert jeder Prozessor diese Teilbäume durch erneute Ausführung

```
second_phase ( )
 i := MyProcID;
 while i < number_stored_nodes do
   copy ( node[i], node[i].g ) to local nodes;
   i := i + nProcs;
 increment (threshold);
 for all local nodes do
   IDA_store ( node, node.g );
```

Abbildung 5.3: Die zweite Phase von *AIDA**

der Routine `IDA_store`. Die Anzahl der gespeicherten Knoten im Array am Ende der zweiten Phase ist abhängig von der Anzahl der in der ersten Phase gespeicherten Knoten und der Zahl der Prozessoren `nProcs`. Für größere Prozessorzahlen ist es sinnvoll, in der zweiten Phase mehrere Iterationen durchzuführen, damit die Zahl der von einem Prozessor gehaltenen Knoten am Ende dieser Phase nicht zu klein ist.

Die Wahl der Konstanten `MAX_NODES` in der ersten Phase und die Zahl der ausgeführten Iterationen in der zweiten Phase von *AIDA** ist von großer Bedeutung für die Effizienz des Algorithmus. Hierdurch wird die Granularität, d. h. die Größe der Teilbäume in der dritten Phase, und die Güte der statischen Lastverteilung am Ende der zweiten Phase bestimmt.

Bis auf das Verbreiten der Wurzel im System ist in den ersten beiden Phasen keinerlei Kommunikation oder Synchronisation notwendig. Jeder Prozessor, der die zweite Phase beendet hat, kann sofort mit der dritten Phase beginnen.

5.4 Die dritte Phase: Dynamischer Lastausgleich am Ende jeder Iteration

In der dritten Phase werden die letzten Iterationen bis zur Expansion eines Lösungsknotens durchgeführt. Dieses erfolgt durch die Anwendung der rekursiven Tiefensuchroutine des sequentiellen Algorithmus *IDA** auf die Teilbäume unter den Knoten der gespeicherten Suchfront. Hat ein Prozessor alle lokal gespeicherten Knoten bis zur gegebenen Kostenschranke `threshold` expandiert, schickt er eine Anfrage an einen Nachbarprozessor. Falls dieser Prozessor noch genügend unbearbeitete Knoten hält, schickt er ein Paket mit unbearbeiteten Knoten an den Absender der Anfrage zurück, andernfalls leitet er die Anfrage an einen anderen Prozessor weiter. Falls sie den Absender wieder erreicht, so konnte kein Prozessor in dieser Iteration unbearbeitete Knoten an ihn abgeben. In diesem Fall beginnt dieser Prozessor direkt mit der

```
IDA ( node, g )
 for all successors succ of node do
   if f ( node.succ ) ≤ threshold
      then if not solved
           then IDA ( node.succ, g(node.succ) );
           else output solution_path and terminate;

third_phase ( )
 while not solved do
   for all local nodes do
      IDA ( node, node.g );
   send work request;
   while new work received do
      add received nodes to local nodes;
      for all new nodes do
           IDA ( node, node.g );
      send work request;
   increment ( threshold );
```

Abbildung 5.4: Die dritte Phase von *AIDA**

nächsten Iteration (eine genaue Beschreibung der Lastverteilungsstrategie in Kapitel 6.3.2).

Da die Prozessoren im System asynchron an verschiedenen Iterationen arbeiten können, ist jede im System verschickte Nachricht mit dem Wert der Variablen `threshold` des Absenders versehen, so daß diese auch nur von Prozessoren ausgewertet wird, die sich in der gleichen Iteration wie der Absender befinden.

Knoten, die ein Prozessor an einen anderen verschickt, werden aus dessen Array gelöscht und stehen dem Empfänger für die folgenden Iterationen zur Verfügung. Hierdurch verringert sich bei jeder Iteration eine eventuell vorhandene Unausgeglichenheit der Initialverteilung (genauere Untersuchungen hierzu findet man in 6.3.2).

5.5 Verbesserungen von AIDA*

Der Algorithmus *AIDA** kann an einigen Stellen noch optimiert werden:

- Zu den einzelnen Teilbäumen können Informationen gesammelt werden, welche für eine Sortierung zwischen den Iterationen von Nutzen sind.

- Nach der ersten Phase können Knoten, welche redundante Zustände repräsentieren, aus dem lokalen Array entfernt werden.

- Die sequentielle Phase, welche eine gute Initialverteilung liefert, könnte durch eine parallele Verteilungsphase simuliert werden.

Zu jedem Teilbaum kann der minimale Wert der heuristischen Funktion gespeichert werden, der in diesem Teilbaum innerhalb einer Iteration aufgetreten ist. Bevor mit der folgenden Iteration begonnen wird, könnten die Arrayelemente nach diesen Werten in aufsteigender Reihenfolge sortiert werden. Dieses hätte zur Folge, daß diejenigen Teilbäume, welche Knoten enthalten, die in der aktuellen Iteration schon nahe an einem Lösungsknoten lagen, in der nächsten Iteration zuerst durchsucht werden. Dieses dürfte die Zahl der betrachteten Knoten in der Lösungsiteration verringern. Diese Sortierung wird in der in Kapitel 6 beschriebenen Implementierung nicht durchgeführt, da die minimalen Werte der heuristischen Funktion in den einzelnen Teilbäumen für das Puzzle-Problem nicht stark voneinander abweichen. Stattdessen wird in der Implementierung am Ende jeder Iteration eine Teilsortierung durchgeführt, wo in einem Lauf über das Array die Elemente, deren Teilbäume eine im Vergleich zu allen Teilbäumen in dieser Iteration durchschnittliche Größe haben, an das Ende des Arrays verschoben werden. Diese Knoten werden in der nächsten Iteration eventuell zur Lastverteilung verschickt. Deshalb ist es sinnvoll, daß die entsprechenden Teilbäume nicht zu klein sind, da ansonsten das Verschicken teurer ist als die Expansion dieser Bäume. Sind die Teilbäume sehr groß, so wird der Zeitraum, in dem die Prozessoren asynchron an unterschiedlichen Iterationen arbeiten, zu groß.

Die am Ende der ersten Phase gespeicherte Suchfront enthält beim Puzzle-Problem im Schnitt etwa 25% redundante Zustände. Diese können am Ende dieser Phase entfernt werden. Wichtig ist, daß derjenige Knoten, der von den jeweiligen Duplikaten am höchsten im Baum liegt, im Array bleibt. Dieses hat jedoch zur Folge, daß die totale Knotenersparnis nach der Eliminierung der Duplikate nur noch etwa 10% beträgt. Zur Ermittlung der redundanten Knoten müssen die Arrayelemente sortiert werden. Dieses Sortieren verlängert die Laufzeit der sequentiellen Phase. Für das 15-Puzzle – im Gegensatz zum 19-Puzzle – ist die Eliminierung der redundanten Knoten aufgrund der kurzen Laufzeiten nicht lohnend. Beim Floorplanning existieren im Suchbaum keine redundanten Zustände.

Eine Parallelisierung der ersten Phase wurde nicht untersucht, da diese Simulation wahrscheinlich sehr kommunikationsintensiv wäre und keine so gute Initialverteilung liefern kann wie die sequentielle Phase.

Kapitel 6

Die Implementierung von AIDA*

Der Algorithmus *AIDA** wurde auf dem Transputersystem *Parsytec GCel* des *Paderborn Center for Parallel Computing (PC^2)* in der Programmiersprache *C* unter dem Betriebssystem *PARIX* implementiert. Mithilfe dieser Implementierung wurden die 100 Instanzen des 15-Puzzles aus [Korf85] auf Netzen mit bis zu 1024 Prozessoren gelöst. Neben der Suche nach der ersten optimalen Lösung *(first-solution-case)* wurden zur Vermeidung von Speedup-Anomalien überwiegend Messungen zur Suche nach allen optimalen Lösungen *(all-solutions-case)* durchgeführt. In diesem Kapitel wird zunächst das System beschrieben, auf dem die Implementierung erfolgte. Danach folgt eine Erklärung der verwendeten Lastverteilungsstrategie auf der Ring- und der Torus-Topologie. Die Ergebnisse für das 15–Puzzle zeigen die Vorteile dieses Algorithmus und lassen auf eine gute Skalierbarkeit auch für größere Systeme schließen. Die Overhead-Funktion wird anhand der bei der Implementierung erreichten Laufzeiten ermittelt. Als weitere Probleme werden einige Instanzen des 19–Puzzles gelöst.

6.1 Der Parsytec GCel und das Betriebssystem PARIX

Der *Parsytec GCel* des *PC*^{2} ist ein 1024-Prozessor Transputer-System. Die *Inmos T805* Prozessoren sind in einem 32×32-Gitter fest verdrahtet, die Taktfrequenz der *RISC*-Prozessoren beträgt 30 MHz. Die Kommunikationsleistung über die Links zweier im Gitter benachbarter Prozessoren beträgt unter dem Betriebssystem *PARIX* etwa 1.1 MByte/sec.

Das Betriebssystem *PARIX* (**PAR**allel extensions to Un**IX**) ermöglicht die Entwicklung von skalierbaren parallelen Programmen. Die Kommunikation erfolgt über virtuelle Links, die durch das Betriebssystem realisiert werden. Dadurch ist Grad eines Prozeßgraphen nicht durch die vier Links eines einzelnen Prozessors beschränkt. Es existieren Bibliotheksfunktionen, welche optimale Einbettungen einer Vielzahl von unterschiedlichen Prozeßgraphen (Ring, 2D-Torus, 3D-Torus, Hypercube, Star, Clique) als virtuelle Topologie auf die Gitterstruktur des *GCel* beschreiben.

Der Algorithmus *AIDA** wurde auf der Ring- und der Torus-Topologie (2D-Torus) implementiert. Der Torus wird in der Regel mit einer Kantenstreckung von 2 auf das Gitter abgebildet. Dieses hat zur Folge, daß die Kommunikation zwischen zwei Nachbarprozessoren im virtuellen Torus über zwei Links im Gitter realisiert wird. Da die Kommunikation im *GCel* nicht über ein spezielles *Routing-Netzwerk* erfolgt, geht die für das Verschicken von Nachrichten benötigte Rechenleistung von der Rechenleistung der – auch indirekt, d. h. aufgrund der Einbettung der virtuellen Links – an der Kommunikation beteiligten Prozessoren ab. Dieses erhöht den Kommunikationsoverhead eines unter *PARIX* implementierten Algorithmus zusätzlich.

6.2 Das Prozeßmodell

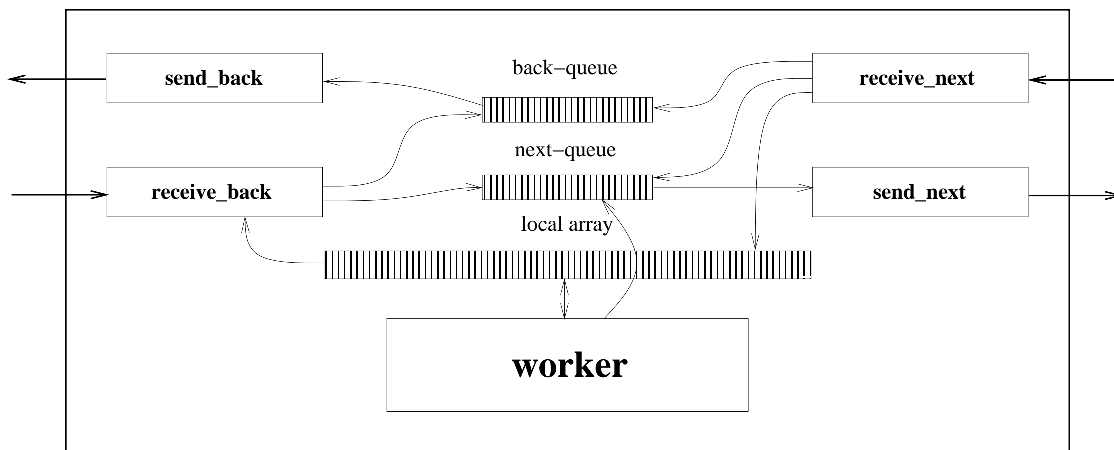


Abbildung 6.1: Das Prozeßmodell für die Implementierung von *AIDA** auf dem Ring

Auf jedem Prozessor laufen 5 (Ring-Topologie) bzw. 9 Prozesse (*Threads*) parallel:

- ein Worker-Thread, der den in Kapitel 5.4 aufgeführten Code ausführt, d. h. die Funktion `IDA` für die Elemente im lokalen Array dieses Prozessors aufruft und nach der Abarbeitung durch Verschicken einer Nachricht neue Arbeit anfordert
- ein Sender-Thread für jeden Kanal, der ein Paket aus einem Puffer verschickt, falls dieser nicht leer ist
- ein Empfänger-Thread für jeden Kanal, der die über den Kanal eingehenden Pakete bearbeitet, d. h. neue Knoten in das lokale Array einfügt oder die Nachricht in einen Sender-Puffer einreiht

Abbildung 6.1 verdeutlicht das Prozeßmodell für die Implementierung auf dem Ring. Für die Implementierung auf dem Torus kommen für jeden zusätzlichen Kanal jeweils ein Sender-Thread, Empfänger-Thread und ein Puffer hinzu.

Es wurden bei der Implementierung die synchronen *PARIX*-Kommunikationsroutinen *Send()* und *Recv()* verwendet. Durch die Sender- und Empfänger-Threads mit den entsprechenden Puffern wird jedoch eine asynchrone Kommunikation unter den einzelnen Prozessoren realisiert. Die Verwendung von synchroner Kommunikation ist aufgrund der sich auf Ringen ergebenen Deadlockgefahr nicht möglich. Die eigene Implementierung der asynchronen Kommunikation anstelle der Benutzung der *PARIX*-Bibliotheks-Routinen *ASend()* und *ARecv()* ermöglicht eine deutlich effizientere Kommunikation.

6.3 Das Lastverteilungsverfahren

6.3.1 Die initiale Lastverteilung in den ersten beiden Phasen

Parallele Algorithmen beinhalten eine initiale Verteilungsphase, in der das Problem soweit aufgesplittet wird, daß jeder Prozessor mit Arbeit versorgt wird. Bei Baumsuchalgorithmen besteht diese Phase aus einer Zerlegung des Suchbaumes in disjunkte Teilbäume, die dann von den einzelnen Prozessoren parallel durchsucht werden.

Die Initialverteilung im Algorithmus *PIDA** (Beschreibung in Kapitel 4.2.1) geschieht durch das gleiche Verfahren, welches auch zur dynamischen Lastverteilung während der Suche verwendet wird. Dieses Arbeitsanziehungsverfahren ist für Netze mit kleinem Knotengrad und großem Durchmesser – insbesondere für die initiale Verteilung – ungeeinet, wie die Berechnung der Isoeffizienz-Funktion für diesen Algorithmus zeigt (Kapitel 7.1.2).

Ein anderes Verfahren beschreiben Korf, Powley und Ferguson in [KorfPowleyFerguson90]. Dieses in ihrem Algorithmus *SIDA** [PowleyFergusonKorf93] benutzte und als *Breadth-First-Allocation-Scheme* bezeichnete Verfahren ist für eine schnelle Initialverteilung ebenfalls ungeeignet. Hierbei wird der Suchbaum in Form einer Breitensuche expandiert, wobei sich die Suchfront mit wachsender Tiefe immer weiter im System verteilt, bis alle Prozessoren einen Knoten des Suchbaumes zur weiteren Expansion erhalten haben. Obwohl dieses Verfahren für die Initialverteilung besser geeignet scheint als das im Algorithmus *PIDA** von Kumar und Rao verwendete Verfahren, so ist doch die Anwendung auf einem massiv parallelen MIMD-System nicht zu empfehlen. Die Zeit für eine Knotenexpansion – insbesondere beim n×n-Puzzle – beträgt in solchen Systemen nur einen Bruchteil der Kommunikationszeit für die Übertragung eines Knotens zwischen zwei benachbarten Prozessoren.

Aus den genannten Gründen wurde für die Initialverteilung im Algorithmus *AIDA** ein vollkommen anderer Ansatz gewählt, der keinerlei Kommunikation – abgesehen vom Verbreiten der Wurzel des Suchbaumes – erfordert. Hierzu wird in der ersten Phase derselbe Baum redundant von allen Prozessoren entwickelt, so daß am Ende alle Prozessoren dieselbe Suchfront in ihren lokalen Arrays halten. Diese Suchfront besteht in der Implementierung im Durchschnitt aus 50 000 Knoten. Zu Beginn der zweiten Phase findet eine Abbildung dieser Suchfront auf alle Prozessoren im System statt, indem jeder Prozessor „seinen" Anteil aus der Suchfront auswählt. Dadurch, daß die Suchfront breit über das System verteilt wird (Prozessor i erhält die Knoten $i, p+i, 2 \cdot p+i \ldots$ aus dem Array), ergibt sich eine viel ausgewogenere Initialverteilung als etwa im *Breadth-First-Allocation-Scheme* von Korf, Powley und Ferguson. In ihrem Verfahren „schiebt" sich die Suchfront durch das System, d. h. die relative Lage der Knoten in der Suchfront entspricht auch der relativen Lage der Knoten im System.

Nach der Aufteilung der Suchfront hält ein einzelner Prozessor bis zu 50 Knoten in seinem lokalen Array. Da in der dritten Phase die dynamische Lastverteilung über die durch die Knoten im Array der Prozessoren beschriebenen Teilbäume erfolgt, muß die Aufteilung des Suchbaumes in disjunkte Teilbäume noch feiner sein. Zu diesem Zweck werden die Knoten in der zweiten Phase weiter expandiert, so daß im System letztendlich eine tiefere und somit größere Suchfront gehalten wird. Für das 15–Puzzle besteht diese Suchfront aus bis zu 3 000 000 Knoten, was in einem System mit 1024 Prozessoren einer Zahl von ca. 3000 Knoten je Prozessor entspricht.

6.3.2 Die dynamische Lastverteilung in der dritten Phase

Nach der zweiten Phase hält jeder Prozessor etwa die gleiche Anzahl von Knoten in seinem lokalen Array. Diese Knoten sind die Wurzeln der Teilbäume, die in der dritten Phase expandiert werden. Dieses geschieht durch Anwendung der rekursiven Tiefensuchroutine aus dem sequentiellen Algorithmus *IDA** innerhalb des `worker`-Threads (siehe Prozeßmodell in Kapitel 6.2). Hat der `worker` alle Knoten im Array des Prozessors abgearbeitet, fügt er eine `GET_WORK`-Nachricht in einen Sender-Puffer ein und blockiert auf einem Semaphor. Die Nachricht bewegt sich zum nächsten Prozessor auf dem Ring. Falls in dessen lokalem Array nicht mehr genügend – d. h. weniger als zwei – unbearbeitete Knoten enthalten sind, wird die Nachricht durch den Empfänger-Thread dieses Prozessors in den Sender-Puffer eingefügt und somit an den nächsten Prozessor im Ring weitergeleitet.

Falls ein Prozessor noch mehr als zwei unbearbeitete Knoten in seinem Array hält, so werden durch dessen `receive`-Thread die Hälfte dieser Knoten, maximal jedoch fünf Arrayelemente, in einem `WORK`-Paket gebündelt und in der entgegengesetzten Richtung auf dem Ring zurückgeschickt. Erreicht diese Antwort den Absender der `GET_WORK`-Nachricht, so werden die neuen Knoten an das Ende des lokalen Arrays angehängt. Der `worker`-Thread wird durch eine Inkrement-Operation auf das Semaphor wieder aktiviert und durchsucht die durch die neuen

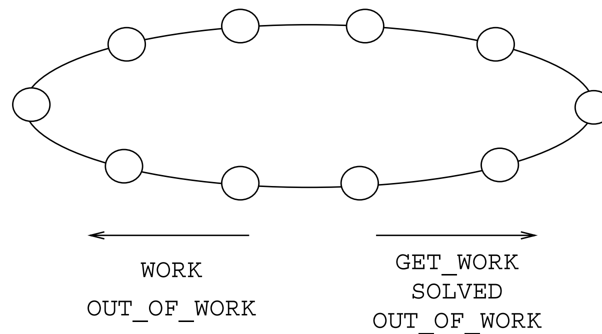


Abbildung 6.2: Der Verlauf der Nachrichten bei der Implementierung auf dem Ring

Knoten beschriebenen Teilbäume.

Falls kein Prozessor auf dem Ring unbearbeitete Knoten in seinem Array hält, so erreicht die GET_WORK-Nachricht wieder ihren Absender. Dieser kann sofort mit der nächsten Iteration beginnen, er versendet jedoch vorher eine OUT_OF_WORK-Nachricht in beide Richtungen, um alle Prozessoren auf dem Ring darüber zu informieren, daß für die aktuelle Iteration keine unbearbeiteten Knoten mehr existieren. Ein Prozessor, der diese OUT_OF_WORK-Nachricht erhalten hat, kann, nachdem er alle Knoten in seinem lokalen Array abgearbeitet hat, direkt mit der nächsten Iteration beginnen, ohne zuvor eine GET_WORK-Nachricht zu verschicken.

Obwohl dieses Lastverteilungsverfahren auch auf größeren Ringen funktioniert, so hat es dort jedoch den Nachteil, daß die Nachrichten gegen Ende der Lastverteilung weitere Wege zurücklegen, da die GET_WORK-Messages nicht mehr von Prozessoren in der näheren Umgebung beantwortet werden. Dieses zeigte sich im Verlauf des Speedups für die Implementierung auf dem Ring [ReinefeldSchnecke94a], der für größere Netze stärker abfällt als bei der Implementierung auf dem Torus. Die Ergebnisse der Implementierung auf dem Ring werden in dieser Arbeit jedoch nicht aufgeführt.

Im zweidimensionalen Torus mit $p = n^2$ Prozessoren liegt jeder Prozessor auf genau zwei Ringen der Länge n, einem vertikalen und einem horizontalen Ring. Auf jedem dieser Ringe läuft das oben beschriebene Lastverteilungsverfahren ab. Nach der Abarbeitung aller Knoten in seinem lokalen Array verschickt ein Prozessor eine GET_WORK-Nachricht auf dem horizontalen Ring. Erreicht ihn diese Nachricht wieder, so verschickt er eine GET_WORK-Nachricht auf seinem vertikalen Ring. Erst wenn auf diesem Ring auch keine unbearbeiteten Knoten vorhanden sind, beginnt der Prozessor mit der nächsten Iteration.

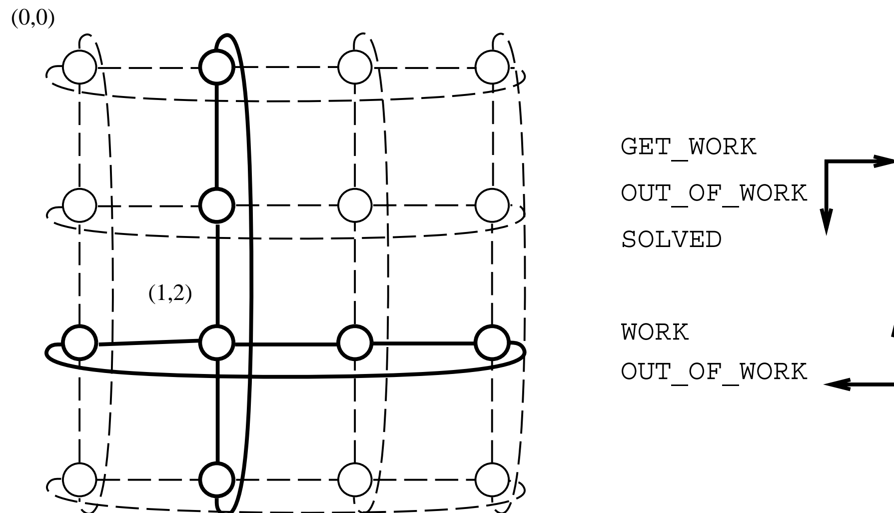


Abbildung 6.3: Der Verlauf der Nachrichten bei der Implementierung auf dem Torus und die von Prozessor (1,2) bei der Lastverteilung berücksichtigten Prozessoren

Die Nachrichten zur Lastverteilung laufen beim $n \times n$-Torus über zwei Ringe der Länge n, und nicht über n^2 Prozessoren wie beim Ring mit $p = n^2$ Prozessoren. Da ein vertikaler und ein horizontaler Ring jeweils nur einen Prozessor gemeinsam haben berücksichtigen alle Prozessoren bei der Lastverteilung unterschiedliche Teilmengen mit $2 \cdot n - 1$ Elementen aus den n^2 Prozessoren des gesamten Systems. Hierdurch ergibt sich eine einfache, aber sehr effektive dynamische Lastverteilung. Eine in der zweiten Phase eventuell entstandene Überlast baut sich schnell ab und wird dabei weit über das System verteilt.

Die von einem Prozessor während der Lastverteilung abgegebenen Knoten werden aus dessen Array gelöscht und in das Array des Empfängers eingetragen. Somit hat dieser Prozessor in den folgenden Iterationen mehr Elemente in seinem Array als zu Beginn der dritten Phase. Das relative Wachstum der Teilbäume zwischen den Iteration ist beim 15-Puzzle weitgehend konstant, so daß sich eine in der Initialverteilung entstandene ungleichmäßige Lastverteilung nach mehreren Iterationen selbstständig ausgleicht. Zu erkennen ist dieses in Abbildung 6.4. Die Zahl der Nachrichten, die durch einen Prozessor im 1024-Torus wandern, nimmt in den letzten Iterationen rapide ab. Dargestellt sind die Durchschnittswerte über alle Probleme mit fünf Iterationen in der dritten Phase[1].

[1] Alle hier und im folgenden dargestellten Graphen beziehen sich auf die Läufe für den *all-solutions-case*, d. h. der Suche nach allen optimalen Lösungen, so daß die letzte Iteration vollständig durchgeführt wird.

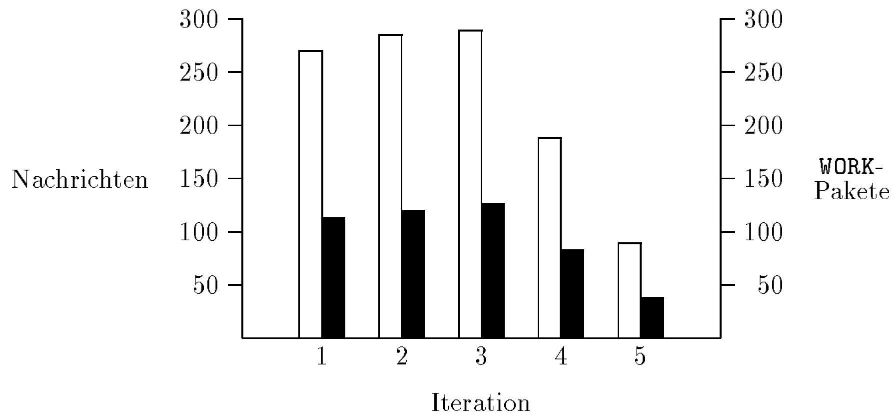


Abbildung 6.4: Die Zahl der Nachrichten pro Prozessor im 32×32-Torus (15-Puzzle)

Ein weiterer Vorteil der Lastverteilungsstrategie von *AIDA** liegt in der Zahl der Nachrichten, die durch einen Prozessor gehen. Diese wächst nicht linear mit der Systemgröße, wie in Abbildung 6.5 deutlich wird, welche die durchschnittliche Zahl der Nachrichten und `WORK`-Pakete zeigt, die in der letzten Iteration durch einen Prozessor gehen. Die dargestellten Kurven flachen für größere Systeme sichtbar ab, was ein noch geringeres Anwachsen der Nachrichten pro Prozessor für Systeme mit mehr als 1024 Prozessoren erwarten läßt. Zu erklären ist dies zum einen dadurch, daß die meisten Anfragen in der direkten Umgebung eines Prozessors beantwortet werden, so daß die mittlere Weglänge der Nachrichten nicht linear mit der Systemgröße wächst. Weiter wächst die Zahl der Prozessoren, welche bei der Lastverteilung von einem Prozessor berücksichtigt werden, nicht linear mit der Systemgröße.

Jeder Prozessor erhält im Algorithmus *AIDA** auch auf größeren Systemen nur wenige (≤ 20) `WORK`-Pakete im Laufe einer Iteration, da die Initialverteilung die Prozessoren schon zu über 90% der Zeit auslastet, ohne das die Last umverteilt werden muß.

6.4 Die Terminierung

Die in Kapitel 4 beschriebenen Parallelisierungen der iterativen Tiefensuche beinhalten eine globale Synchronisation am Ende jeder Iteration. Realisiert wird dieses durch einen Terminierungsalgorithmus wie er in [DijkstraFeijenGastern83] beschrieben wird. Diese Überprüfung, ob alle Prozessoren die Iteration beendet haben, ist sehr kommunikationsaufwendig und trägt stark zum Overhead des entsprechenden Algorithmus bei. Im Algorithmus *AIDA** gibt es zwischen den Iterationen in den parallelen Phasen keine globale Synchronisation. Da durch eine Sortierung (vgl. Kapitel 5.5) am Ende jeder Iteration wird dafür gesorgt wird, daß die Pakete,

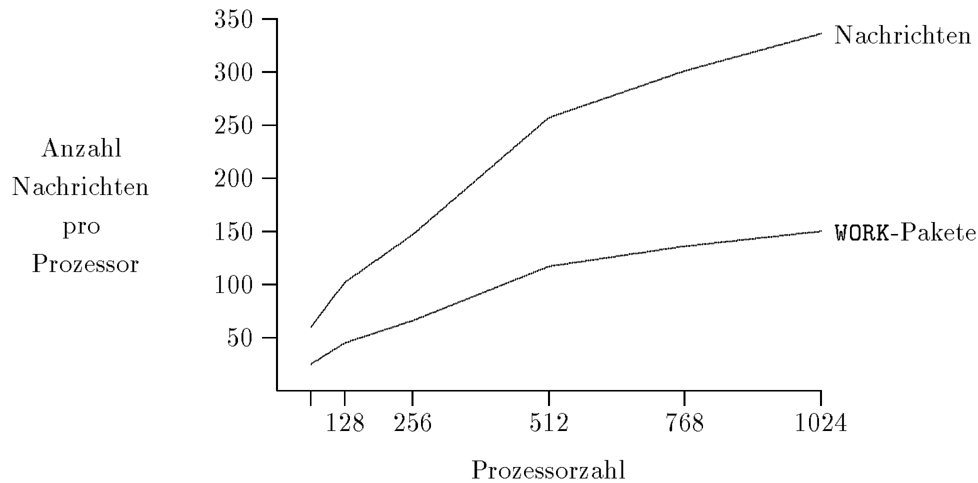


Abbildung 6.5: Die Zahl der Nachrichten pro Prozessor in der letzten Iteration (Probleme 51 bis 100 des 15-Puzzles)

die zur Lastverteilung verwendet werden, eine mittlere Größe haben, ist der Zeitraum, in dem die Prozessoren an unterschiedlichen Iterationen arbeiten, begrenzt. Aufgrund dieser tolerierten Asynchronität ist zwischen den einzelnen Iterationen in der Implementierung des Algorithmus *AIDA** keine Terminierungserkennung – wie etwa das Verfahren nach Dijkstra – notwendig.

Es muß nur sichergestellt werden, daß das Programm terminiert, falls ein Prozessor eine Lösung gefunden hat. Zu diesem Zweck existiert in jedem Prozessor ein `solved`-Flag, welches zusammen mit einer Variablen `solved_threshold` beschreibt, ob und in welcher Iteration eine Lösung gefunden wurde. Hat ein Prozessor einen Lösungsknoten expandiert, so verschickt er eine `SOLVED`-Nachricht auf beiden Ringen. Jeder Prozessor, der diese Nachricht über seinen horizontalen Ring empfängt, setzt seine Variablen und schickt die Nachricht auf seinen beiden Ringen weiter. Ein Prozessor, der diese Nachricht über seinen vertikalen Ring erhält, leitet diese auch nur über den vertikalen Ring weiter[2]. Abbildung 6.6 verdeutlicht das Verbreiten einer Lösung auf dem Torus. Bei der Suche nach der ersten optimalen Lösung *(first-solution-case)* wird das `solved`-Flag innerhalb der rekursiven Tiefensuch-Routine überprüft. Falls die Lösung in der aktuellen Iteration gefunden wurde, so bricht ein Prozessor, dessen Flag gesetzt wurde, die Suche ab und terminiert. Arbeitet der Prozessor noch an einer Iteration mit einer kleineren Kostenschranke, so setzt er die Suche fort, bis er alle Elemente in seinem lokalen Array abgearbeitet hat. Findet dieser Prozessor dabei eine Lösung, die somit besser als die schon ermittelte ist, so wird diese wiederum an alle Prozessoren weitergegeben. Dadurch wird sichergestellt, daß

[2] Dieses Verteilen wird über die `receive`-Threads realisiert, so daß sich die Lösung unabhängig von dem Zustand der `worker`-Threads im System verteilt.

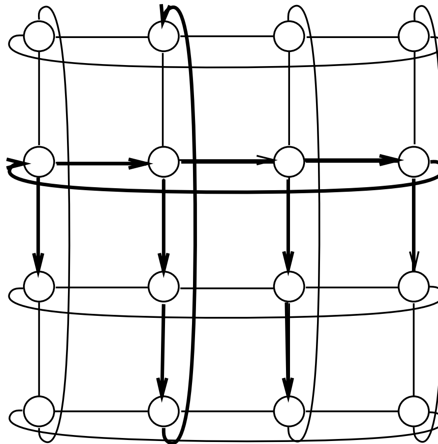


Abbildung 6.6: Das Verbreiten einer Lösung auf der Torus-Topologie

die gefundene Lösung auch optimal ist.

Falls alle optimalen Lösungen gesucht werden, durchsuchen die Prozessoren noch alle durch die Knoten in ihren Arrays beschriebenen Teilbäume bis zur Kostenschranke des Lösungsknotens. Dabei wird auch am Ende der letzten Iteration die dynamische Lastverteilung durchgeführt. Prozessoren, die mit einer größeren Kostenschranke als der des Lösungsknotens arbeiten brechen in beiden Fällen sofort die Suche ab.

6.5 Auswirkungen der unterschiedlichen Teilbaumgrößen

Problematisch ist – insbesondere bei der Suche nach allen optimalen Lösungen – die unterschiedliche Größe der Teilbäume, die von den Prozessoren in der dritten Phase durchsucht werden. Obwohl alle Knoten der gespeicherten Suchfront denselben Wert der Kostenfunktion haben, ist die Größe dieser Bäume – also die Granularität der Last – sehr unterschiedlich. Abbildung 6.7 zeigt die Größe aller Teilbäume in der letzten Iteration für das 50. Problem des 15-Puzzles bei der Suche nach allen Lösungen. Die Werte stammen aus einem Lauf auf 64 Prozessoren und sind logarithmisch aufgetragen. Die durchschnittliche Teilbaumgröße beträgt 782 Knoten, der Median ist 230. Über 5% der Teilbäume haben mehr als 4000 Knoten. Wie man sieht hat der größte Baum mehr als 300 000 Knoten, während die kleinsten Bäume selbst in der letzten Iteration nur aus 8 Knoten bestehen. Die Expansionszeit für den größten Teilbaum beträgt 8.8 Sekunden. Die durchschnittliche Laufzeit der letzten Iteration dieses Problems sind 58 Sekunden, d. h. derjenige Prozessor, welcher den größten Teilbaum hält, ist während der letzten Iteration zu 15% mit der Expansion dieses Baumes beschäftigt. Liegt der Baum am Ende des Arrays, so ist

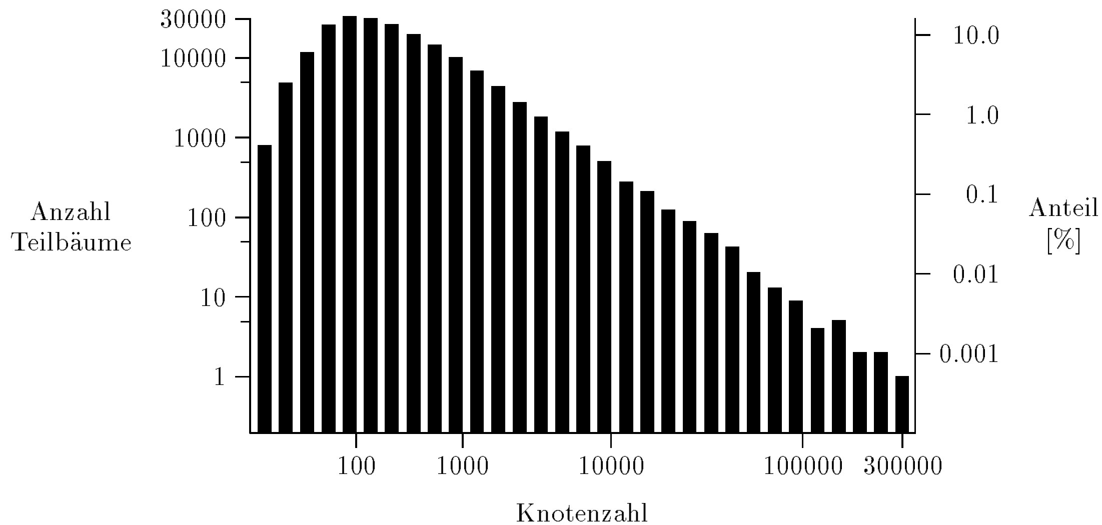


Abbildung 6.7: Die Größe aller Teilbäume in der letzten Iteration, logarithmische Skalierung (15-Puzzle, all-solutions-case)

der Prozessor noch maximal 8.8 Sekunden mit der Suche in diesem Baum ausgelastet und kann davon keine Last mehr abgeben. Dieser Fall wirkt sich deutlich auf den Speedup aus (nähere Untersuchungen in Kapitel 6.6.2). Um diesen Effekt zu verringern, werden am Ende jeder Iteration in der dritten Phase die Knoten im Array eines Prozessors nach der aktuellen Teilbaumgröße sortiert[3], so daß am Ende des Arrays nur nahezu durchschnittlich große Lastpakete stehen, über die in der nächsten Iteration die Lastverteilung realisiert wird. Wie in Abbildung 6.8 zu sehen ist, ist die Differenz zwischen den Terminierungszeiten der Prozessoren nach Anwendung dieses Verfahrens annehmbar. Außerdem ist die Suche nach allen Lösungen in der Praxis nicht von so großem Interesse, und die unterschiedlichen Teilbaumgrößen zeigen bei der Suche nach der ersten Lösung keine direkte Wirkung.

6.6 Die Ergebnisse für das Puzzle

Die Implementierung von *AIDA** wurde mit den 100 Instanzen des 15-Puzzles aus [Korf85] und einigen Instanzen des 19-Puzzles getestet. Zur Vermeidung von Speedup-Anomalien (siehe Kapitel 3.1) wurde in erster Linie der *all-solutions-case*, also die Suche nach allen optimalen Lösungen untersucht, es liegen aber auch Messungen für die Suche nach der ersten optimalen Lösung vor.

[3] Es handelt sich um keine vollständige Sortierung, es werden lediglich bei einem Lauf über das Array einige ungünstige Knoten vertauscht.

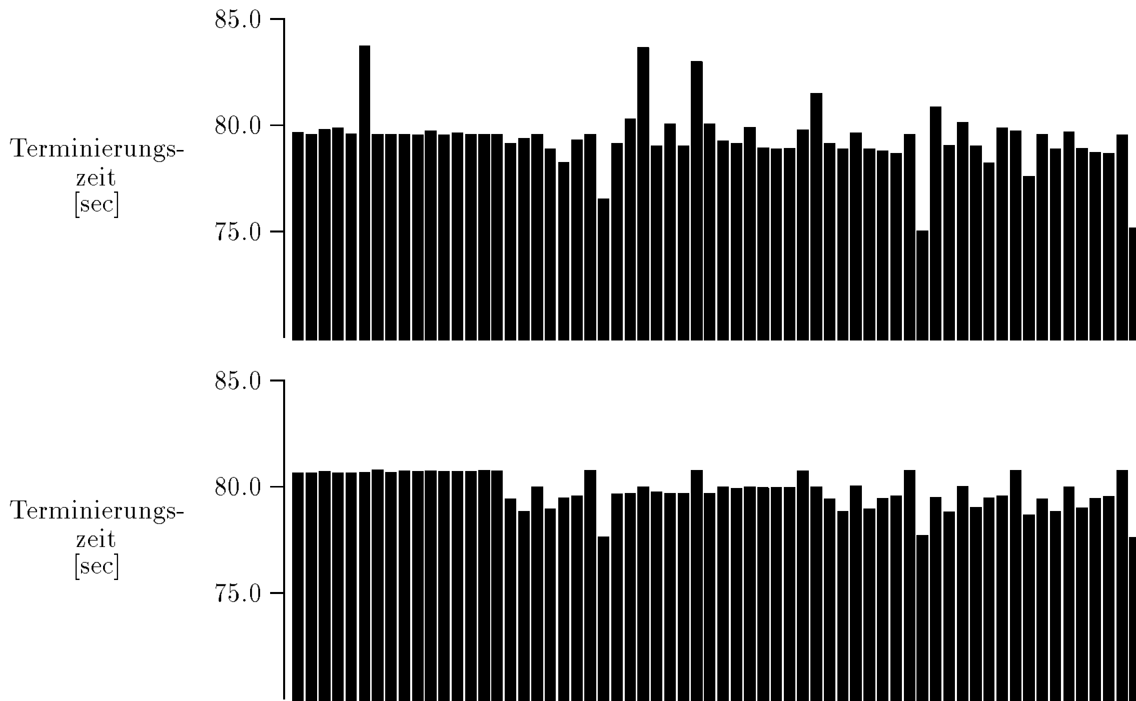


Abbildung 6.8: Die Terminierungszeiten der 64 Prozessoren ohne (oben) und mit (unten) Sortierung zwischen den Iterationen (15-Puzzle, Problem 50, all-solutions-case)

Zur Speedupberechnung wurden die Laufzeiten einer optimalen sequentiellen Implementierung des Algorithmus *IDA** für das Puzzle auf einem T805 Prozessor ermittelt, es wurde also nicht der aufwendige Code des Algorithmus *AIDA** auf einem Prozessor ausgeführt. Diese Implementierung expandiert etwa 35000 Knoten in einer Sekunde im Falle des 15-Puzzles und etwa 41000 Knoten in der Sekunde bei der Lösung des 19-Puzzles.

Die mittlere sequentielle Laufzeit für eines der 100 Probleme des 15-Puzzles aus [Korf85] beträgt für die Suche nach allen Lösungen 678.4 Minuten, der Median liegt bei 96.3 Minuten. Es ergibt sich somit eine Gesamtlaufzeit von 1130.7 Stunden (47.1 Tagen) für alle 100 Instanzen. Dem gegenüber steht eine Laufzeit von 5047.3 Sekunden, also 84.1 Minuten auf 1024 Prozessoren, was einen Speedup von 806.5 und eine Effizienz von 78.8% bedeutet, wenn man alle 100 Instanzen als ein Problem betrachtet.

Für die Suche nach der ersten optimalen Lösung der 100 Instanzen des 15-Puzzles ergibt sich eine mittlere sequentielle Laufzeit von 170.1 Minuten, was eine Gesamtlaufzeit von 283.5 Stunden bzw. 11.8 Tagen für alle 100 Probleme bedeutet. Die Gesamtlaufzeit auf 1024 Prozes-

soren beträgt 24.2 Minuten. Der sich so errechnende Speedup für die Lösung aller 100 Instanzen beträgt 703.8, die Effizienz 68.7%.

Es wurden 13 kleinere Instanzen des 19-Puzzles (4×5-Puzzle) durch die Implementierung des Algorithmus *AIDA** gelöst. Dabei wurde auf 1024 Prozessoren im Mittel ein Speedup von 835.0 erreicht, was einer Effizienz von 81.5% entspricht. Die mittlere Laufzeit auf 1024 Prozessoren beträgt 29.5 Minuten.

Klasse	Knotenzahl	MD $h(r)$	Lösungstiefe $h*(r)$	seq. Laufzeit [sec]	
				Mittelw.	Median
I	9017002	34.8	47.1	250	171
II	72048774	37.1	51.9	2284	1823
III	344114162	36.0	54.0	9820	7769
IV	5260859214	40.3	59.2	150463	62726
I – IV	1421609788	37.1	53.1	40704	5780

Tabelle 6.1: Die vier Problemklassen für das 15-Puzzle (Werte für *all-solutions-case*)

6.6.1 Das 15-Puzzle

Für die Speedupbetrachtung wurden die 100 Probleme des 15-Puzzles in vier Klassen zu je 25 Instanzen eingeteilt, wobei die Probleme dazu nach ihrer Größe, d. h. der Zahl der bei der Suche nach allen Lösungen expandierten Knoten sortiert wurden. Die mittleren Werte für die Instanzen aus den vier Klassen sind in Tabelle 6.1 aufgelistet. Die entsprechenden Werte für die einzelnen Probleme sind im Anhang (Tabellen A.1 bis A.4, Seiten 78 bis 81) zu finden.

Bei der Suche nach der ersten Lösung treten Speedup-Anomalien auf. Um einen anomaliefreien Speedup zu erhalten, muß die Leistung des parallelen Baumsuchalgorithmus unabhängig von der Knotenzahl gemessen werden. Ein solches Maß liefert der in Kapitel 3.1 eingeführte normierte Speedup:

$$sp_{norm}(p) = \frac{T_{seq} \cdot w_{AIDA^*}(p)}{w_{IDA^*} \cdot T_{par}(p)}$$

Die Tabelle 6.2 zeigt die Ergebnisse für die Suche nach der ersten optimalen Lösung. In den folgenden Tabellen stellen die Laufzeiten bzw. Speedups der einzelnen Klassen die Durchschnittswerte über die 25 Probleme in einer Klasse dar, die angegebenen Speedup-Werte stehen also

Klasse	I	II	III	IV	I – IV
$T_{par}(64)$	2.8	14.1	62.4	752.1	207.8
$sp(64)$	34.4	53.3	54.1	56.9	49.7
$sp_{norm}(64)$	32.8	47.9	52.8	54.1	46.9
$T_{par}(128)$	1.8	8.0	31.2	295.7	84.2
$sp(128)$	52.8	92.7	106.0	118.7	92.6
$sp_{norm}(128)$	73.1	84.5	100.6	106.8	91.3
$T_{par}(256)$	1.7	4.8	17.1	145.7	42.3
$sp(256)$	59.1	151.6	195.2	244.1	162.6
$sp_{norm}(256)$	176.1	140.2	187.8	206.2	177.6
$T_{par}(512)$	2.0	3.8	9.5	71.6	21.7
$sp(512)$	55.6	215.8	324.0	462.2	264.4
$sp_{norm}(512)$	191.4	221.5	318.5	392.3	280.9
$T_{par}(768)$	2.8	4.1	7.9	52.9	16.9
$sp(768)$	46.5	196.7	411.3	634.0	322.2
$sp_{norm}(768)$	341.7	294.2	423.4	549.0	402.1
$T_{par}(1024)$	3.6	4.3	7.1	43.1	14.5
$sp(1024)$	39.0	181.4	458.4	749.7	357.1
$sp_{norm}(1024)$	501.4	351.2	488.8	706.9	512.1

Tabelle 6.2: Die Ergebnisse bei der Suche nach der ersten Lösung (15-Puzzle)

nicht in direkter Relation zu den entsprechenden Laufzeiten!

Wie man sieht ist für die Probleme der Klasse IV der durchschnittliche normierte Speedup kleiner als der allgemeine Speedup. Das zeigt, daß in dieser Klasse ein überlinearer Speedup überwiegt. Dies kann z. B. daran liegen, daß die Instanzen in dieser Klasse mehrere optimale Lösungen besitzen.

Im folgenden wird nun die Suche nach allen optimalen Lösungen betrachtet, was die Leistungsanalyse erleichtert, da in diesem Fall der sequentielle Algorithmus *IDA** und der Algorithmus *AIDA** bei der Suche exakt die gleichen Knoten expandieren.

Abbildung 6.9 zeigt die erreichten Speedups auf der Torus-Topologie mit bis zu 1024 Prozessoren für die Suche nach allen Lösungen. Wie man sieht, sind die Speedup-Kurven für die kleineren Probleme deutlich schlechter, da die durchschnittlichen Laufzeiten hier sehr kurz sind

Klasse	$T_{par}(64)$	$T_{par}(128)$	$T_{par}(256)$	$T_{par}(512)$	$T_{par}(768)$	$T_{par}(1024)$
I	6.4	4.0	2.9	3.0	4.0	5.6
II	34.3	19.5	10.6	7.4	6.7	6.9
III	158.3	80.5	44.1	23.9	18.0	15.5
IV	2429.6	1228.2	608.6	318.1	225.4	173.9
I – IV	657.2	333.1	166.6	88.1	63.5	50.5

Tabelle 6.3: Die mittleren Laufzeiten (in Sekunden) der vier Klassen des 15-Puzzles *(all-solutions-case)*

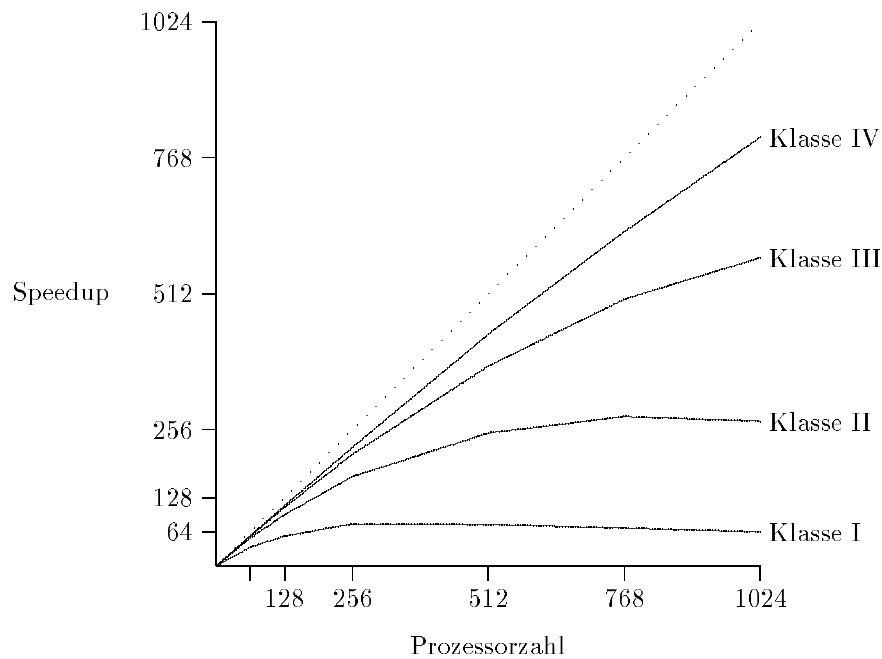


Abbildung 6.9: Die Speedup-Graphen für die vier Klassen des 15-Puzzles *(all-solutions-case)*

(Klasse I 5.6 Sekunden bzw. Klasse II 6.9 Sekunden auf 1024 Prozessoren) und sich somit die erste Phase und der Kommunikationsoverhead stärker auswirken. Genauere Überlegungen zu den Overheads in *AIDA** werden in Kapitel 6.6.2 angestellt. Die Tabelle 6.3 zeigt die mittleren parallelen Laufzeiten für die Probleme in den vier Klassen. Die Laufzeit der ersten Phase von *AIDA** beträgt bei 1024 Prozessoren im Schnitt 2.95 Sekunden, was bei den Problemen in den Klassen I und II schon 52.7% bzw. 42.8% der Gesamtlaufzeit ausmacht. Aus diesem Grund

werden bei einer genaueren Analyse der Overheads nur die Klassen III und IV berücksichtigt.

6.6.2 Die Overheads

Es existieren im Algorithmus *AIDA** eine Reihe von Faktoren, welche sich negativ auf den Speedup auswirken:

- Die erste sequentielle Phase ist für eine gute Initialverteilung notwendig. Die Dauer dieser Phase ist in der Implementierung abhängig von der Anzahl der Prozessoren. Damit die Auswirkungen dieses sequentiellen Anteils auf den Speedup gering bleiben, muß die parallele Laufzeit – also das Problem – eine bestimmte Größe haben.

- Die zur Berechnung der Speedups herangezogenen sequentiellen Laufzeiten wurde mit einer optimalen Implementierung des Algorithmus *IDA** ermittelt[4]. Der Code des Algorithmus *AIDA** ist – nicht zuletzt aufgrund der Kommunikations-Threads – deutlich umfangreicher.

- Die Verzögerung beim Verbreiten der Lösung im *first-solution-case* und die sich aufgrund der ungleichmäßigen Teilbaumgrößen ergebene Terminierungsphase im *all-solutions-case* führen zu einer Verlängerung der eigentlichen parallelen Gesamtlaufzeit und wirken sich somit negativ auf den Speedup aus.

In den Abbildungen 6.10 und 6.11 sind unterschiedliche Speedupgraphen für die einzelnen Klassen dargestellt, wobei den Werten die Zeiten für die Suche nach allen Lösungen zugrundeliegen. Dabei wurden die folgenden Overheads berücksichtigt:

S_{opt}: Dieser Graph stellt den maximal möglichen Speedup dar, d. h. als einziger Overhead wird nur die Dauer der ersten Phase berücksichtigt, für die Parallelisierung der anderen Phasen wird ein optimaler Speedup vorausgesetzt. Diese Kurve beschreibt zwar nur einen hypothetischen Speedup, zeigt aber nochmals – insbesondere in Abbildung 6.11 – daß der durch die sequentielle Phase bestimmte Overhead erst in großen parallelen Systemen relevant wird, wo die Laufzeit der ersten Phase besonders groß ist, während die Gesamtlaufzeit sehr kurz wird.

S_{real}: Dieser Speedup berücksichtigt alle Overheads bis auf die Terminierung. Ein Speedup in dieser Größenordung ist für die Suche nach der ersten optimalen Lösung (ohne Speedup-Anomalien) zu erwarten und wurde auch erreicht (vgl. Tabelle 6.2). Zu beachten ist lediglich, daß die Probleme bei der Suche nach allen Lösungen deutlich größer sind, so daß die Auswirkung der ersten Phase geringer ist.

[4]Die Ausführung von *AIDA** auf einem Prozessor ist aufgrund des begrenzten Speichers ohnehin nicht möglich, da die in der dritten Phase von *AIDA** gehaltene Suchfront die Speichergröße eines einzelnen Prozessors bei weitem übersteigt.

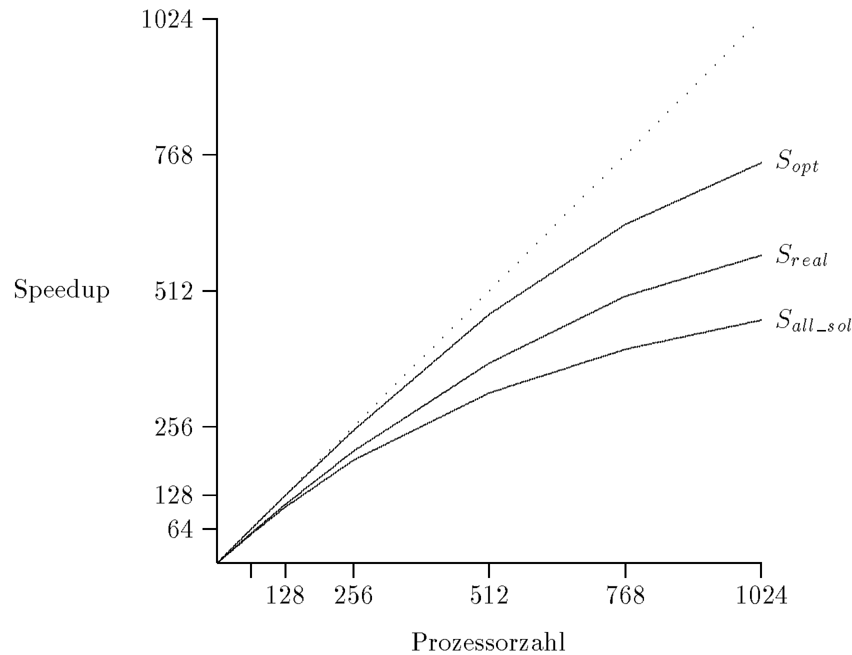


Abbildung 6.10: Speedup-Graphen für das 15-Puzzle (Klasse III)

S_{all_sol}: In diesem Fall wird bei der Speedupberechnung die Laufzeit desjenigen Prozessors benutzt, der als letzter die Suche beendet. Aufgrund der in Kapitel 6.7 beschriebenen unterschiedlichen Größe der Teilbäume in der letzten Iteration kann die Laufzeit dieses Prozessors mehrere Sekunden über der durchschnittlichen Laufzeit aller Prozessoren im System liegen und somit den Speedup sehr stark negativ beeinflussen. In der Praxis ist die Suche nach der ersten Lösung üblich, bei der dieser Overhead nicht auftritt.

Die Laufzeit der ersten Phase ist in der Implementierung von *AIDA** unabhängig von der Problemgröße und wird abhängig von der Prozessorzahl so gewählt, daß nach dieser Phase eine Suchfront in jedem Prozessor gehalten wird, die groß genug ist, um alle Prozessoren mit Knoten zu versorgen. Die Laufzeit der ersten Phase ist linear abhängig von der Prozessorzahl, die Formel für die absolute Laufzeit (in Sekunden) der ersten Phase in der Implementierung von *AIDA** lautet:

$$T_{1st\,Phase}(p) = 0.003 \cdot p$$

Wie in Abbildung 6.12 deutlich wird, beträgt der Anteil der Laufzeit der ersten Phase an der Gesamtlaufzeit auf 1024 Prozessoren für ein Problem der Klasse IV im Schnitt weniger als 2.0% und für ein Problem der Klasse III etwa 10.0%. Da die mittleren Laufzeiten der Probleme aus den Klassen I und II 5.7 bzw. 10.5 Sekunden sind, werden 51.4% bzw. 27.8% der Laufzeit für die

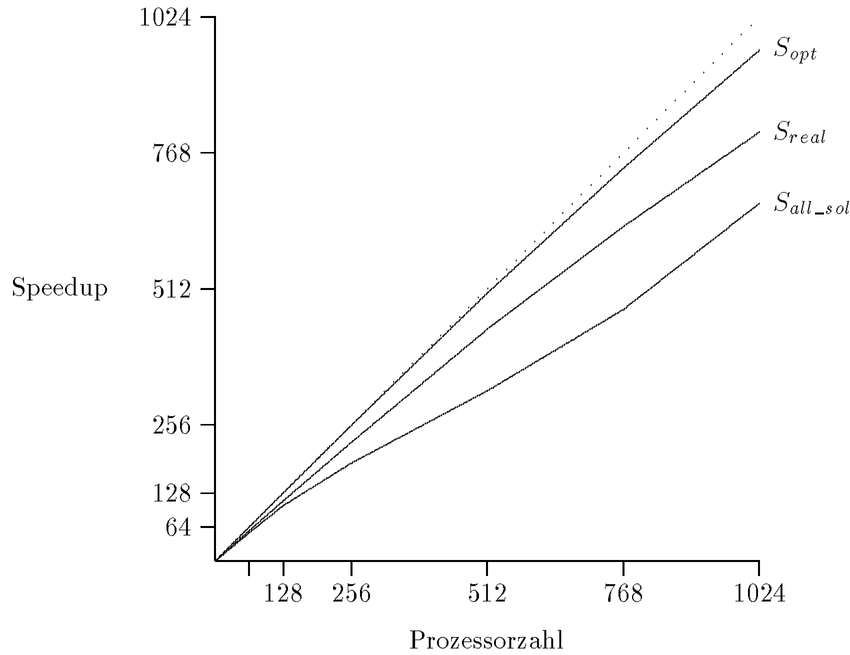


Abbildung 6.11: Speedup-Graphen für das 15-Puzzle (Klasse IV)

Initialverteilung in der ersten Phase verwendet. Für die Berechnung einer (theoretischen) oberen Schranke nach Amdahls Gesetz (siehe Kapitel 3.2) mit dieser sequentiellen Phase muß der Anteil der ersten Phase an der Gesamtlaufzeit ermittelt werden. Dieser Anteil liegt zwischen 0.003% (Klasse IV) und 1.2% (Klasse I). Abbildung 6.13 zeigt den nach Amdahl möglichen Speedup auf 1024 Prozessoren mit einer sequentiellen ersten Phase von 2.95 Sekunden.

Die Differenz zwischen den Kurven S_{opt} und S_{real} in den Abbildungen 6.10 und 6.11 beschreibt den Restoverhead des Algorithmus *AIDA**. Dieser wird durch den Mehraufwand der parallelen Version gegenüber der zur Bestimmung der sequentiellen Laufzeit herangezogenen Implementierung des Algorithmus *IDA** für das 15-Puzzle bestimmt. Den wesentlichen Anteil macht dabei der Kommunikationsoverhead aus. Er setzt sich zusammen aus der CPU-Zeit der Kommunikationsthreads (neben dem `worker`-Thread laufen bei der Implementierung auf der Torus-Topologie noch jeweils vier Sender- und vier Empfänger-Threads, vgl. Kapitel 6.2), der Idle-Zeit der Prozessoren während einer Arbeitsanfrage und der Zeit für das Durchrouten von Nachrichten durch das Betriebssystem *PARIX* (wegen der virtuellen Topologien nehmen die Nachrichten andere Wege als die in den Kommunikationsroutinen von *AIDA** beschriebenen, vgl. Kapitel 6.1). Dieser Overhead wurde in Kapitel 3.4 mit $T_o(p)$ bezeichnet, da er in der Regel abhängig von der Prozessorzahl p ist. Es wurde die folgende Formel für die Laufzeit eines

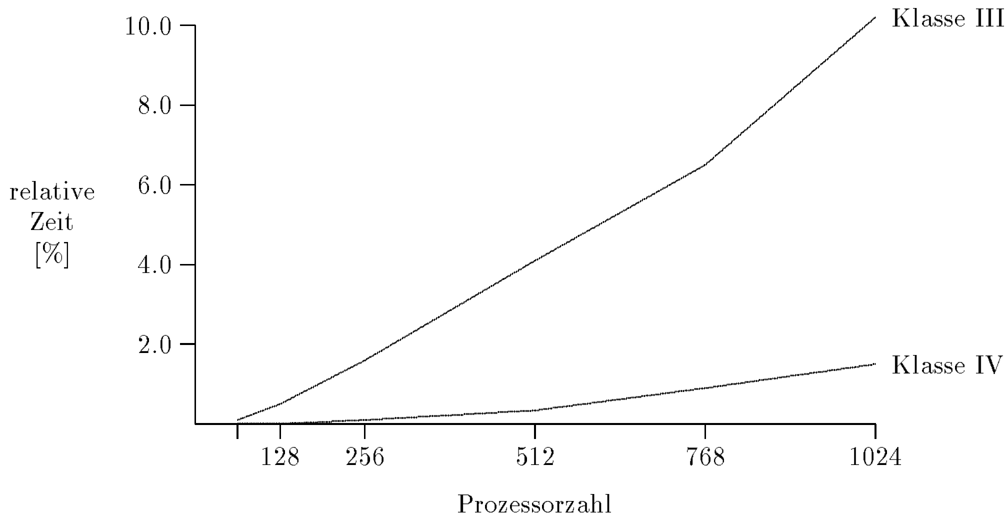


Abbildung 6.12: Relativer Anteil der ersten Phase an der Gesamtlaufzeit *(all-solutions-case)*

parallelen Algorithmus aufgestellt:

$$T_{par}(p) = T_{seq} \cdot \mathcal{S} + \frac{\mathcal{P} \cdot T_{seq}}{p} + \frac{T_o(p)}{p}$$

Interessant ist nun der Overheadanteil an der parallelen Laufzeit, also die Overheadzeit je Prozessor $\frac{T_o(p)}{p}$ dividiert durch die parallele Laufzeit $T_{par}(p)$:

$$\frac{T_o(p)}{p \cdot T_{par}(p)} = 1 - \frac{T_{seq} \cdot \mathcal{S}}{T_{par}(p)} - \frac{\mathcal{P} \cdot T_{seq}}{p \cdot T_{par}(p)}$$

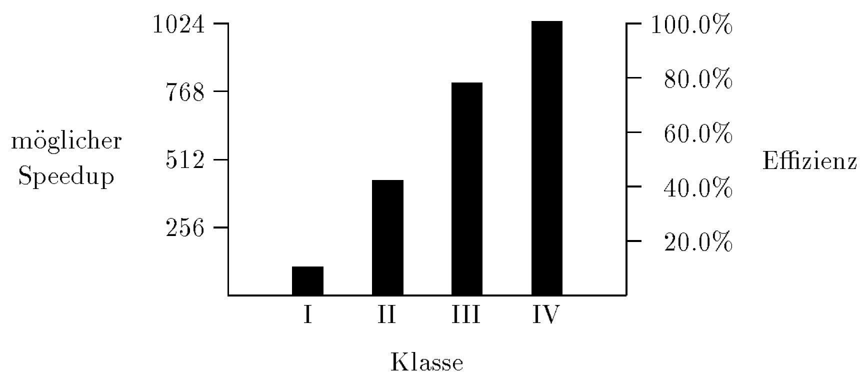


Abbildung 6.13: Der nach Amdahls Gesetz erreichbare Speedup für die vier Klassen des 15-Puzzles auf 1024 Prozessoren

Da $\mathcal{S}$ der nichtparallelisierbare Anteil des sequentiellen Algorithmus ist, gilt

$$\mathcal{S} = \frac{T_{1st\,Phase}(p)}{T_{seq}}$$

und somit für den parallelisierbaren Anteil $\mathcal{P}$:

$$\mathcal{P} = 1 - \mathcal{S} = 1 - \frac{T_{1st\,Phase}(p)}{T_{seq}}$$

Somit ergibt sich als Formel für den Overheadanteil:

$$\begin{aligned}
\frac{T_o(p)}{p \cdot T_{par}(p)} &= 1 - \frac{T_{1st\,Phase}(p)}{T_{par}(p)} - \frac{T_{seq}}{p \cdot T_{par}(p)} + \frac{T_{1st\,Phase}(p)}{p \cdot T_{par}(p)} \\
&= 1 - \frac{T_{1st\,Phase}(p)}{T_{par}(p)} - \frac{T_{seq} - T_{1st\,Phase}(p)}{p \cdot T_{par}(p)} \\
&= 1 - \frac{p \cdot T_{1st\,Phase}(p) + T_{seq} - T_{1st\,Phase}(p)}{p \cdot T_{par}(p)} \\
&= 1 - \frac{T_{1st\,Phase}(p) \cdot (p-1) + T_{seq}}{p \cdot T_{par}(p)}
\end{aligned}$$

Setzt man die oben bestimmte Formel für die Laufzeit der ersten Phase ein, so erhält man:

$$\frac{T_o(p)}{p \cdot T_{par}(p)} = 1 - \frac{0.003 \cdot p \cdot (p-1) + T_{seq}}{p \cdot T_{par}(p)}$$

Diese Formel dient nun zur Berechnung des Overheadanteils an den gemessenen parallelen Laufzeiten. Tabelle 6.4 zeigt diesen Overhead-Anteil für die Klassen III und IV.

p	$T_o(p)$ [sec.]		$\frac{T_o(p)}{p \cdot T_{par}(p)}$ [%]	
	Klasse III	Klasse IV	Klasse III	Klasse IV
64	299	5019	2.9	3.2
128	435	6698	4.2	4.2
256	1273	5143	11.3	3.3
512	1632	11619	13.3	7.2
768	2237	20877	16.2	12.1
1024	2909	24467	18.3	13.7

Tabelle 6.4: Die Gesamtzeit des Restoverheads und der Anteil an der Laufzeit

Die Gesamtzeit des Restoverheads wächst linear in der Prozessorzahl, für die Probleme in Klasse III gilt $T_o(p) \approx 3 \cdot p$ und für die Probleme in Klasse IV gilt $T_o(p) \approx 24 \cdot p$.

6.6.3 Das 19-Puzzle

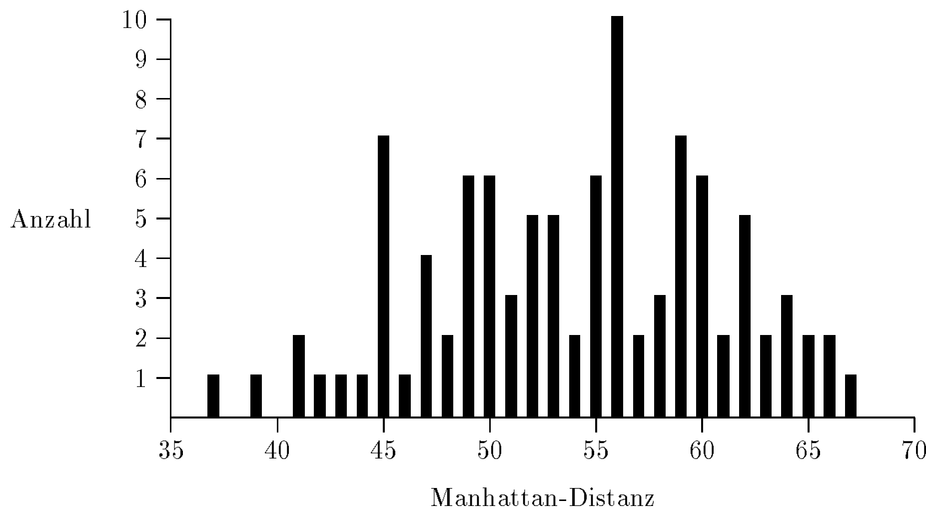


Abbildung 6.14: Die Manhattan-Distanz der 100 zufällig erzeugten Instanzen des 19-Puzzles

Das 19-Puzzle ist eine nicht quadratische Variante des $n \times n$-Puzzles mit fünf Zeilen und vier Spalten. Da hier – im Gegensatz zu den 100 Instanzen des 15-Puzzles von Korf – keine Reihe von bekannten Problemen existiert, wurden 100 zufällige Instanzen generiert. Abbildung 6.14 zeigt die Verteilung der Manhattan-Distanz der Ausgangsstellungen dieser Puzzles. Die 100 Instanzen wurden nach dieser Manhattan-Distanz sortiert und 21 kleinere Probleme sequentiell gelöst. Die Lösung erfolgte auf einer *SPARCstation 10*, wobei die Laufzeit für das größte gelöste Problem auf diesem Rechner mehr als drei Wochen beträgt. In diesem Fall wurden bei der Suche über $1.5 \cdot 10^{12}$ Knoten expandiert.

In Tabelle A.5 im Anhang (Seite 82) sind die 13 mittelgroßen Instanzen aufgeführt, welche durch den Algorithmus *AIDA** gelöst wurden. Die im sequentiellen Fall auf einem *T805 Transputer* erreichte Zahl von 41000 Knotenexpansionen in der Sekunde liegt deutlich höher als beim 15-Puzzle. Diese Arbeitsrate wurde anhand von 5 kleinen Instanzen ermittelt, welche sequentiell auf einem *T805* Transputer gelöst wurden. Es ist jedoch anzunehmen, daß die Arbeitsrate für die größeren Instanzen niedriger ist, da der in Tabelle 6.5 aufgeführte normierte Speedup, dem eine sequentielle Arbeitsrate von 41000 Knotenexpansionen pro Sekunde zugrundeliegt, für die Größe der Probleme eigentlich zu klein ausfällt[5].

In Tabelle 6.5 sind die Ergebnisse für die 13 durch die Implementierung von *AIDA** gelösten

[5] Die Initialverteilung lastet beim 19-Puzzle die Prozessoren teilweise zu über 99% aus, so daß alle Prozessoren in dieser Zeit die sequentielle Expansionsroutine auf ihre lokalen Teilbäume anwenden.

Nr.	$T_{par}(512)$	$sp_{norm}(512)$	$T_{par}(768)$	$sp_{norm}(768)$	$T_{par}(1024)$	$sp_{norm}(1024)$
1	4826.1	424.8	303.9	634.7	226.9	843.3
2	2543.1	422.3	71.1	615.7	126.9	836.6
3	1558.6	421.6	74.2	572.1	48.0	721.5
4	15649.7	429.0	1063.0	642.3	1036.0	856.7
5	195432.7	431.2	12113.5	633.4	9111.4	861.9
6	2380.4	421.2	106.1	622.8	86.6	827.6
7	13374.7	424.8	880.7	636.8	1119.2	845.8
8	16561.5	423.4	1695.8	633.5	983.9	841.9
9	6087.4	425.3	783.5	630.7	398.0	830.0
10	29878.8	425.0	1906.3	636.9	1186.3	848.1
11	16355.2	423.7	1284.5	632.1	916.3	841.1
12	46912.2	424.6	2772.5	631.3	2198.0	848.0
13	111353.6	427.0	6221.1	636.3	5607.6	852.5
	3560.9	424.9	2252.0	627.6	1772.7	835.0

Tabelle 6.5: Die Laufzeiten (in Sekunden) und Speedups für das 19-Puzzle

Instanzen des 19-Puzzles aufgeführt. Die erreichten Speedups liegen durchweg höher als beim 15-Puzzle, da die Laufzeiten größer sind. Der mittlere Speedup auf 1024 Prozessoren beträgt 835.0, was einer Effizienz von 81.5% entspricht. Den höchsten Speedup von 861.9, also eine Effizienz von 84.2%, erreichte das größte Problem bei einer Laufzeit von 151.9 Minuten.

6.7 Die Floorplan-Optimierung

Obwohl die Floorplan-Optimierung über die typischen Eigenschaften (geringe Lösungsdichte, schlechte Schranken, hoher Verzweigungsfaktor) verfügt, welche die Probleme auszeichnen, die sich mittels iterativer Tiefensuche lösen lassen, wurde im Laufe der Implementierung jedoch deutlich, daß der heuristische Verzweigungsfaktor zu gering ist.

Es wurde die folgende Heuristik verwendet: Jeder innere Knoten n im Suchbaum beschreibt einen Teil-Floorplan. Der Wert $g(n)$ beschreibt die Fläche dieses Teil-Floorplans, d. h. das Produkt der längsten Wege in den Graphen $\mathcal{G}$ und $\mathcal{H}$, welche den Teil-Floorplan repräsentieren (vgl. Kapitel 2.5.2). Der Wert der Knotenbewertungsfunktion $f(n)$ ist das Produkt der längsten Wege in beiden Graphen, wenn diejenigen Kanten hinzugefügt werden, die Blöcke repräsentieren, die noch nicht im entsprechenden Teil-Floorplan enthalten sind. Diese Kanten werden mit

den minimalen Ausmaßen der entsprechenden Blöcke gewichtet.

Problem	Fläche	IDA*			DFSBB
		b	Iterationen	Knoten	Knoten
EX1	121	2.85	10	11 105	23 553
EX2	176	1.58	28	717 942	485 808
EX3	484	1.10	142	21 545 995	1 591 035
EX4	352	1.17	105	467 710 192	55 318 416

Tabelle 6.6: Die Daten der sequentiellen Läufe für die Floorplan-Optimierung

Diese einfache Heuristik liefert für die Probleme EX1 bis EX4 (siehe Anhang Seite 82) einen mittleren heuristischen Verzweigungsfaktor b von 1.675. Für das Problem EX3 liegt dieser Wert nur bei 1.10. Es ergibt sich somit durch die redundanten Expansionen in den Iterationen vor der Lösungsiteration nach Kapitel 2.3 ein Mehraufwand von

$$\frac{W_{IDA^*}}{W_{A^*}} = 121$$

gegenüber einer entsprechenden Implementierung des Algorithmus *A**. Eine Anwendung der Algorithmen *IDA** bzw. *AIDA** auf die Floorplan-Optimierung wäre also nur mit einer besseren Heuristik sinnvoll.

Ein paralleler Branch&Bound-Tiefensuchalgorithmus (DFSBB) mit der oben beschriebenen Heuristik liefert bessere Ergebnisse [ReinefeldSchnecke94a]. Tabelle 6.6 zeigt die in diesem Fall – zumindest für die drei größeren Probleme – deutlich geringeren Knotenzahlen im Gegensatz zur Implementierung des Algorithmus *IDA**.

Aufgrund der im sequentiellen Fall gemessenen hohen Zahl von Iterationen wurde auf eine Implementierung des Algorithmus *AIDA** für das Floorplanning verzichtet. Als weiteres Problem kommt hinzu, daß – im Gegensatz zum n×n–Puzzle – das Inkrement der Kostenschranken zwischen den Iterationen nicht bekannt ist, sondern global ermittelt werden muß. Dieses hat einen hohen Kommunikationsaufwand oder eine globale Synchronisation zwischen den Iterationen zur Folge. Außerdem ist zu beachten, daß selbst das Problem EX3 bei einer erreichten Arbeitsrate von 770 Knoten pro Sekunde bei der Implementierung von *IDA** auf einem *T805* Transputer noch ein zu kleines Problem ist, um ein System mit 1024 Prozessoren auszulasten.

p	64	128	256	512	768
$T_{min}(p)$	552.4	651.8	393.0	196.2	146.4
$T_{max}(p)$	564.5	661.9	414.7	225.3	182.0
$T_{avg}(p)$	557.4	653.3	400.6	204.2	158.5
$T_{load}(p)$	30.9	30.0	23.6	17.8	17.1
Knoten	25 888 260	60 124 935	73 677 455	74 081 900	85 332 685
Knoten/sec	725.7	719.0	718.5	708.5	700.8

Tabelle 6.7: Die Ergebnisse des parallelen Branch&Bound-Algorithmus für die Floorplan-Optimierung

Der Branch&Bound-Algorithmus durchsucht den gleichen Suchbaum wie die oben beschriebene Implementierung des Algorithmus *IDA**. Es werden Tiefensuchen durchgeführt, wobei abgebrochen und die Suche an einer anderen Stelle im Baum fortgesetzt wird, falls die Fläche eines Teil-Floorplans die Größe des bislang kleinsten vollständigen Floorplans übersteigt. Bei der Parallelisierung dieses Algorithmus wurde das gleiche Lastverteilungsverfahren wie im Algorithmus *AIDA** verwendet, zum Verbreiten einer neuen Schranke wurde das in Kapitel 6.4 verwendete Verfahren benutzt. Die Ergebnisse der parallelen Version dieses Branch&Bound-Algorithmus für das Problem EX3 sind in Tabelle 6.7 aufgeführt.

Im Gegensatz zur Implementierung von *AIDA** ist die Dauer der ersten Phase nicht abhängig von der Prozessorzahl. In allen Läufen ist der Suchbaum in 78125 Teilbäume aufgespalten. Dieses erklärt die steigende Differenz der Terminierungszeiten für größere Prozessorzahlen ($T_{min}(p)$, $T_{max}(p)$, $T_{avg}(p)$ bezeichnen die minimale, maximale und durchschnittliche Laufzeit der einzelnen Prozessoren). Wie auch schon bei der Implementierung von *AIDA** zeigt sich daß eine kurze sequentielle Phase – in diesem Fall hat die erste Phase eine Laufzeit von 6.5 Sekunden – eine Initialverteilung liefert, welche die Prozessoren zu 90% der Laufzeit auslastet. Deutlich wird dieses an dem Anteil der Zeit $T_{load}(p)$, in der die Prozessoren nach der Abarbeitung der initial zugewiesenen Knoten neue Arbeit anfordern und die dabei erhaltenen Teilbäume durchsuchen.

Wie man an der Zahl der expandierten Knoten sehen kann, ergibt sich bei der parallelen Version ein extrem großer Suchoverhead. Dieser Suchoverhead läßt sich durch die Verwendung einer besseren Heuristik verringern. Man kann jedoch an der durchschnittlichen Arbeitsrate ablesen, daß auch für diesen Algorithmus der Overhead nur sehr gering ist, da die Arbeitsrate der sequentiellen Implementierung 770 Knoten in der Sekunde beträgt. Hieraus ergibt sich ein normierter Speedup von 699 für 768 Prozessoren, was einer Effizienz von 91% entspricht. Im Gegensatz zum Puzzle-Problem kann man in diesem Fall jedoch kaum von einem Speedup

sprechen, da bei dessen Berechnung der enorme Suchoverhead unberücksichtigt bleibt.

Kapitel 7

Die Skalierbarkeit von AIDA*

Wie schon deutlich wurde, ist die sequentielle Phase im Algorithmus *AIDA** vertretbar, falls die Probleme eine bestimmte Größe haben. Ein wesentlicher Vorteil der Initialverteilung in *AIDA** ist, daß dazu keinerlei Kommunikation nötig ist. In MIMD-Systemen ohne gemeinsamen Speicher ist die Zeit für die Kommunikation eines Knotens im Vergleich zur Expansionszeit sehr teuer, insbesondere für die in dieser Arbeit betrachteten Probleme. Das von Kumar, Rao und Ramesh in ihrem Algorithmus *PIDA** verwendete Verfahren beinhaltet gerade in der Anfangsphase einen erheblichen Kommunikationsaufwand. Dadurch ergibt sich eine sehr zeitaufwendige und ungleichmäßige Initialverteilung. Deutlich wird dieses anhand der Isoeffizienz-Funktion, welche Kumar und Rao für das Verfahren auf dem Ring und dem Hypercube errechnen. In diesem Kapitel wird außerdem noch die Isoeffizienz-Funktion für den Torus berechnet. Diese wird mit der Isoeffizienz-Funktion für den Algorithmus AIDA* verglichen, wobei deutlich wird, daß dieser Algorithmus aufgrund des geringen Kommunikationsoverheads im Gegensatz zum Verfahren von Kumar und Rao auf einem Torus besser skalierbar ist.

7.1 Die Skalierbarkeit des Algorithmus PIDA*

7.1.1 Die Isoeffizienz-Funktion für den Ring

Kumar und Rao berechnen in [KumarRao90] die Isoeffizienz-Funktion für ihre parallele Version der iterativen Tiefensuche auf dem Ring und dem Hypercube. In ihrem Verfahren wird diese Funktion maßgeblich durch die Kommunikationszeit zur Lastverteilung bestimmt, wobei insbesondere die initiale, gleichmäßige Verteilung der Arbeit über alle Prozessoren auf Netzen mit kleinem konstanten Grad – wie z. B. dem Ring – sehr zeitaufwendig ist.

Es gelten die folgenden Definitionen:

w	Gesamtarbeit, Knotenzahl des gesamten Suchbaumes
p	Prozessorzahl

P_i	i-ter Prozessor im Ring
$trans(p)$	Anzahl der Stacktransfers für p Prozessoren
T_{comm}	Laufzeit für einen Stacktransfer zwischen zwei benachbarten Prozessoren

Wie in Kapitel 4.2.1 beschrieben, hält zunächst nur Prozessor P_0 die gesamte Arbeit w in Form der Wurzel des Suchbaumes. Er expandiert deren Nachfolger und legt sie auf dem Stack ab. Bei Erreichen einer bestimmten Tiefe wird der Stack, d. h. die Arbeit w, in zwei Teile der Größe $\alpha \cdot w$ und $(1-\alpha) \cdot w$ geteilt, wobei $\alpha \cdot w$ der kleinere Teil der Arbeit ist, d. h. es gilt $0 \leq \alpha < 0.5$. Der Stack der Größe $(1-\alpha) \cdot w$ wird an den Prozessor P_1 weitergegeben, der diesen weiterentwickelt und dann wiederum einen Teil des neuen Stacks an den nächsten Prozessor weitergibt.

Ein Prozessor P_i im Ring erhält somit zunächst einen Anteil von $(1-\alpha)^i \cdot w$ der Gesamtarbeit. Um den Ring gut auszulasten, sollte jeder Prozessor einen Arbeitsanteil von $\frac{w}{p}$ erhalten. Hierzu sind für diesen Prozessor $\frac{w/p}{(1-\alpha)^i \cdot w}$ Stack-Transfers notwendig. Daraus kann man nun die minimale Anzahl der Stack-Transfers berechnen:

$$trans(p) = \sum_{i=0}^{p-1} \frac{1}{p \cdot (1-\alpha)^i} = \frac{1}{p} \cdot \sum_{i=0}^{p-1} \beta^i = \frac{1}{p} \cdot \frac{\beta^p - 1}{\beta - 1} \quad \text{mit} \quad \beta := \frac{1}{1-\alpha}$$

Als untere Schranke für den Gesamtoverhead gilt die Gesamtkommunikationszeit, welche sich direkt aus der berechneten Anzahl der nötigen Stacktransfers ergibt:

$$T_o(p) \geq T_{comm} \cdot \frac{1}{p} \cdot \frac{\beta^p - 1}{\beta - 1}$$

Nach Kapitel 3.4 gilt für die Isoeffizienz-Funktion:

$$W(p) = \Omega\left(\frac{E}{1-E} \cdot T_o(p)\right)$$

Somit ergibt sich die folgende Isoeffizienz-Funktion für den Algorithmus *PIDA** bei der Implementierung auf dem Ring:

$$\begin{aligned} W(p) &= \Omega\left(\frac{E}{1-E} \cdot \frac{T_{comm} \cdot (\beta^p - 1)}{p \cdot (\beta - 1)}\right) \\ \Longrightarrow W(p) &= \Omega\left(\frac{\beta^p}{p}\right) \qquad \text{mit} \qquad \beta := \frac{1}{1-\alpha} \end{aligned}$$

Diese Isoeffizienz-Funktion besagt, daß die Größe des Suchbaumes bei diesem Verfahren auf dem Ring exponentiell mit der Prozessorzahl wachsen muß, um eine konstante Effizienz zu gewährleisten. Somit ist eine Skalierbarkeit auf der Ring-Topologie nicht gegeben.

Also Isoeffizienz-Funktion für den Hypercube ermitteln Kumar und Rao

$$W(p) = \Omega\left(p^{\log\frac{1+\beta}{2}}\right)$$

mit $\beta := \frac{1}{1-\alpha}$ wie auch schon bei der Isoeffizienz für den Ring.

Für $\beta > 3$ muß die Zahl der Knoten im Suchbaum nur polynomiell mit der Zahl der Prozessoren wachsen, für $\beta < 3$ ergibt sich eine sublineare Isoeffizienz-Kurve. Kumar und Rao verweisen für diesen Fall darauf, daß es sich bei der angegebenen Isoeffizienz-Funktion nur um eine untere Schranke handelt. In ihren empirischen Untersuchungen ermittelten sie als realistische Isoeffizienz-Funktion für den Hypercube

$$W(p) \sim c \cdot p^{1.59} \qquad (c \text{ konst.},\ 1.59 = \log 3,\ \beta \approx 5).$$

Diese – im Gegensatz zum Ring – recht günstige Isoeffizienz-Funktion für den Hypercube ist aufgrund des großen, nicht konstanten Knotengrades nicht weiter verwunderlich. Da jedoch ein Hypercube mit größeren Prozessorzahlen ($\geq 2^{10}$) technisch nur sehr schwer zu realisieren ist, ist es sinnvoll die Isoeffizienz-Funktion des Verfahrens von Kumar und Rao auch für andere Netze zu berechnen. Für Netze mit konstantem Knotengrad (z. B. CCC, DeBruijn, Gitter, Torus) dürfte die Isoeffizienz-Funktion ähnlich ungünstig wie für den Ring aussehen.

7.1.2 Die Isoeffizienz-Funktion für den Torus

Zu Beginn hält wieder nur ein beliebiger Prozessor P_0 im Torus die Wurzel des Suchbaumes. Ein Prozessor P_d, der den Abstand d von diesem Prozessor hat, erhält zunächst nur einen Arbeitsanteil von $(1-\alpha)^d \cdot w$. Um eine ausgewogene Arbeitsverteilung zu erlangen, müßte dieser Prozessor $\frac{w/p}{(1-\alpha)^d \cdot w}$ Stack-Transfers erhalten. Analog zum Ring kann man nun die Zahl der Stack-Transfers im Torus abschätzen, wobei die Zahl der Knoten mit Abstand d von Prozessor P_0 $4 \cdot d$ beträgt:

$$\begin{aligned}
trans(p) &\geq \sum_{d=1}^{\sqrt{p}} \frac{4 \cdot d}{(1-\alpha)^d \cdot p} \\
&= \frac{4}{p} \cdot \sum_{d=1}^{\sqrt{p}} \frac{d}{(1-\alpha)^d} \\
&= \frac{4}{p} \cdot \frac{\sum_{d=1}^{\sqrt{p}} d \cdot (1-\alpha)^{\sqrt{p}-d}}{(1-\alpha)^{\sqrt{p}}} \\
&\geq \frac{4}{p} \cdot \frac{\sqrt{p}}{(1-\alpha)^{\sqrt{p}}}
\end{aligned}$$

$$= \frac{4}{\sqrt{p} \cdot (1-\alpha)^{\sqrt{p}}}$$

$$\Longrightarrow trans(p) = \frac{4 \cdot \beta^{\sqrt{p}}}{\sqrt{p}} \quad \text{mit} \quad \beta := \frac{1}{1-\alpha}$$

Somit ergibt sich als eine untere Schranke für den Gesamtoverhead:

$$T_o(p) \geq T_{comm} \cdot \frac{4}{\sqrt{p}} \cdot \beta^{\sqrt{p}}$$

Diese Schranke, eingesetzt in die Formel für die Isoeffizienz-Funktion, ergibt:

$$W(p) = \Omega\left(\frac{E}{1-E} \cdot \frac{T_{comm} \cdot 4 \cdot \beta^{\sqrt{p}}}{\sqrt{p}}\right)$$

$$\Longrightarrow W(p) = \Omega\left(\frac{\beta^{\sqrt{p}}}{\sqrt{p}}\right) \quad \text{mit} \quad \beta := \frac{1}{1-\alpha}$$

Die Isoeffizienz-Funktion beschreibt – wie auch schon für den Ring – eine für eine konstante Effizienz benötigte exponentiell in der Prozessorzahl wachsende Problemgröße. Dieses bestätigt wieder die geringe Skalierbarkeit des Parallelisierungsansatzes von Kumar und Rao auf Netzen mit konstantem Knotengrad.

7.2 Die Auswirkungen der sequentiellen Phase in AIDA*

An den im Kapitel 6 dargestellten Speedupgraphen für das 15-Puzzle (Abbildungen 6.10 und 6.11) zeigt sich, daß der Einfluß der sequentiellen ersten Phase auf den Speedup von *AIDA** relativ gering, falls die Probleme eine bestimmte Größe haben. Im folgenden wird untersucht, wie groß ein Problem sein muß, damit trotz der sequentiellen Anfangsphase die Effizienz nicht zu sehr abfällt.

Sei $T_{par\,Phases}(p)$ die parallele Laufzeit der Phasen 2 und 3 des Algorithmus *AIDA**. Dann gilt für den maximal erreichbaren Speedup[1]:

$$sp(p) = \frac{T_{seq}}{T_{par}(p)} = \frac{T_{1st\,Phase}(p) + p \cdot T_{par\,Phases}(p)}{T_{1st\,Phase}(p) + T_{par\,Phases}(p)}$$

Sei Q das Verhältnis der Laufzeit der ersten Phase zur Laufzeit der parallelen Phasen:

$$Q(p) = \frac{T_{1st\,Phase}(p)}{T_{par\,Phases}(p)}$$

Dann gilt als obere Schranke für den Speedup:

$$sp(p) = \frac{Q(p) + p}{Q(p) + 1}$$

[1] Es wird wieder von einem optimalen Speedup für die Phasen 2 und 3 ausgegangen.

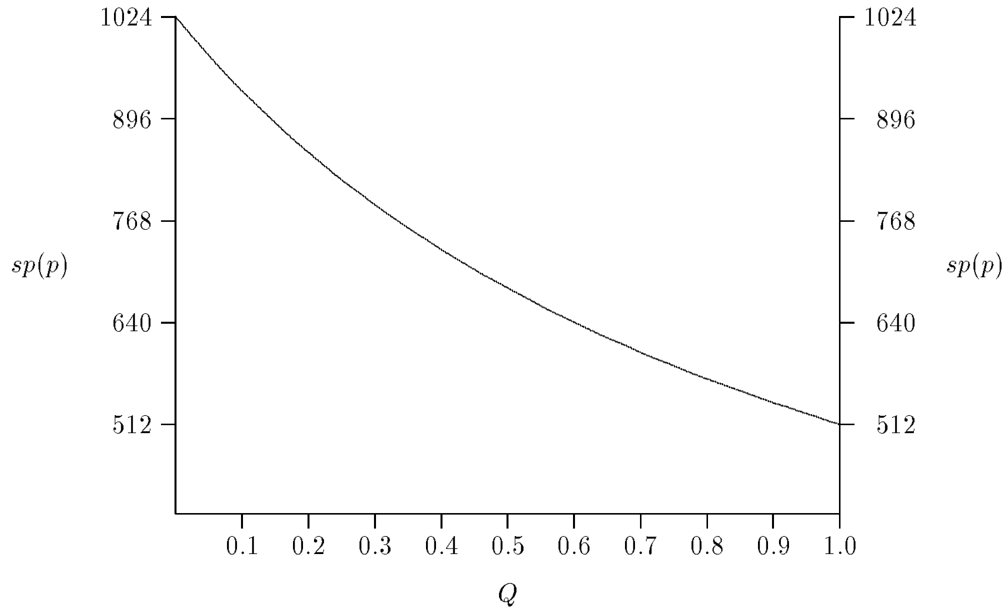


Abbildung 7.1: Der Verlauf des Speedups auf 1024 Prozessoren für das Verhältnis Q aus Laufzeit der ersten Phase zur Laufzeit der beiden parallelen Phasen

Abbildung 7.1 zeigt den Verlauf des Speedups auf 1024 Prozessoren für unterschiedliche Werte von Q.

Für die Effizienz gilt entsprechend:

$$eff(p) = \frac{Q(p) + p}{p \cdot (Q(p) + 1)}$$

Um nun eine Aussage über die Problemgröße, d. h. die parallele bzw. sequentielle Laufzeit zu machen, welche notwendig ist, um eine erste Phase auszugleichen, also um eine bestimmte Effizienz zu gewährleisten, muß die Gleichung für die Effizienz nach $Q(p)$ aufgelöst werden:

$$Q(p) = \frac{p - p \cdot eff(p)}{p \cdot eff(p) - 1}$$

Es ergibt sich also für die Laufzeit der parallelen Phasen bei konstanter angestrebter Effizienz E:

$$T_{par\,Phases}(p) = \frac{T_{1st\,Phase}(p)}{Q(p)} = \frac{T_{1st\,Phase}(p) \cdot (p \cdot E - 1)}{p \cdot (1 - E)}$$

Ausgehend von dieser Formel kann nun bei einer festen Laufzeit für die erste Phase die benötigte sequentielle Gesamtlaufzeit zum Erreichen einer bestimmten Effizienz für *AIDA** auf

$T_{1st\,Phase}(1024)$ [sec]	$eff(1024)$ [%]	$T_{par}(1024)$ [sec]	T_{seq} [min]
0.5	95	10.0	162
	90	5.0	77
	75	2.0	26
	50	1.0	9
1.0	95	20.0	324
	90	10.0	154
	75	4.0	51
	50	2.0	17
2.0	95	40.0	649
	90	20.0	307
	75	8.0	102
	50	4.0	34
3.0	95	60.0	973
	90	30.0	461
	75	12.0	154
	50	6.0	51

Tabelle 7.1: Durch die erste Phase bedingte Gesamtlaufzeiten

1024 Prozessoren berechnet werden. Die Laufzeiten für unterschiedlich lange erste Phasen sind in Tabelle 7.1 aufgeführt. So muß z. B. für eine Effizienz von 95% die Laufzeit auf 1024 Prozessoren 60 Sekunden betragen, um eine sequentielle Phase von 3 Sekunden auszugleichen. Dabei wird allerdings für die beiden anderen Phasen ein optimaler Speedup vorausgesetzt.

7.3 Die Isoeffizienz-Funktion von AIDA*

In Algorithmus *PIDA** wird die Isoeffizienz-Funktion durch die zur Lastverteilung nötige Kommunikation bestimmt. Bedingt durch eine gute initiale Verteilung in der ersten Phase ist dieser Anteil für die Effizienz von *AIDA** nicht von so großer Bedeutung. Insbesondere für große Netze sind die Auswirkungen der Kommunikationszeit im Vergleich zur Laufzeit der ersten Phase nur gering.

Wie in Kapitel 3.4 berechnet, lautet eine allgemeine Formel für die Isoeffizienz-Funktion:

$$W(p) = \Omega\left(\frac{E}{1-E} \cdot ((p-1) \cdot T_s + T_o(p))\right)$$

T_s bezeichnet die Laufzeit des sequentiellen Anteils, d. h. es gilt:

$$T_s = T_{1st\,Phase}(p) = 0.003 \cdot p$$

Der Overhead von AIDA* wächst in der in Kapitel 6 beschriebenen Implementierung linear in der Prozessorzahl, nach Kapitel 6.6.2 gilt:

$$T_o(p) = O(p)$$

Es ergibt sich somit als Isoeffizienz-Funktion für die Implementierung von AIDA*:

$$\begin{aligned} W(p) &= \Omega\left(\frac{E}{1-E} \cdot (0.003 \cdot p \cdot (p-1) + T_o(p))\right) \\ \Longrightarrow W(p) &= \Omega(p^2) \end{aligned}$$

Die Isoeffizienz-Kurve für *AIDA** wird in Abbildung 7.2 gezeigt. Zum direkten Vergleich ist die in Kapitel 7.1.2 berechnete Isoeffizienz-Funktion für den Algorithmus *PIDA** auf dem Torus in dem Graphen dargestellt. Man erkennt die – im Vergleich zu *PIDA** – auch für größere Prozessorzahlen noch verhältnismäßig gering wachsende Isoeffizienz-Kurve für den Algorithmus *AIDA**. In den Kurven wird von einer Effizienz von 95% ausgegangen. Es werden die folgenden Funktionen dargestellt:

PIDA*: $T_{seq} = \frac{E}{1-E} \cdot \frac{T_{comm} \cdot 4 \cdot \beta\sqrt{p}}{\sqrt{p}}$ mit $T_{comm} = 0.12 \cdot 10^{-3}$ $\beta = 2$

AIDA*: $T_{seq} = \frac{E}{1-E} \cdot (T_s \cdot (p-1) + T_o(p))$ mit $T_s = 0.003 \cdot p$ $T_o(p) = 10 \cdot p$

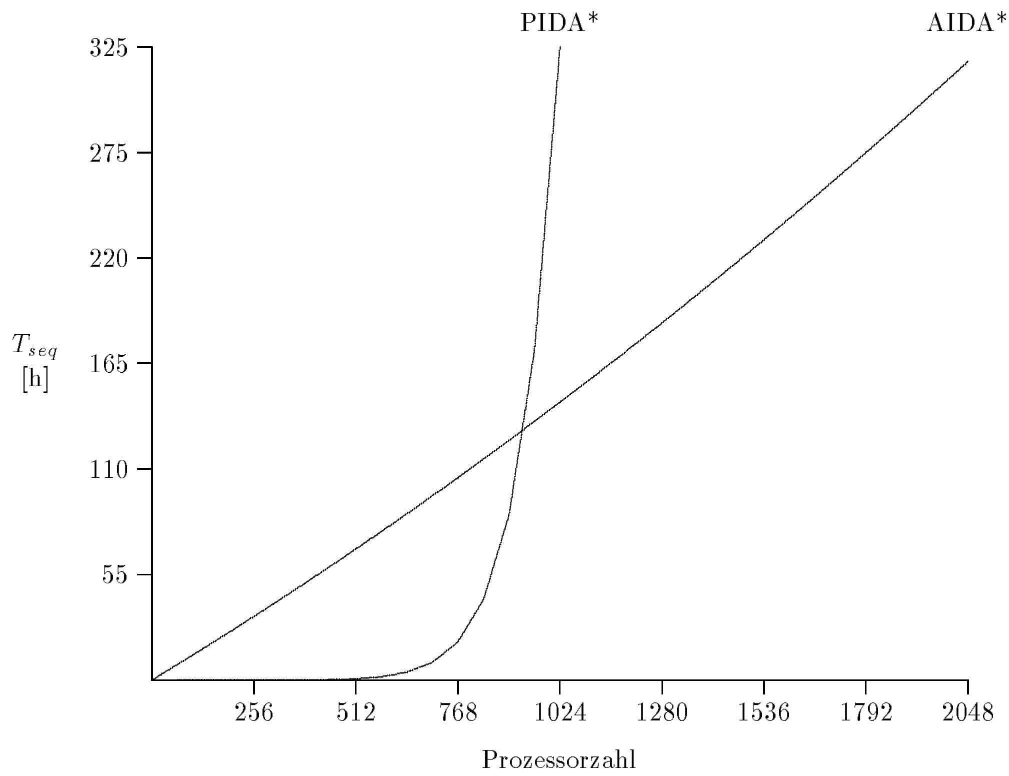


Abbildung 7.2: Die Isoeffizienz-Kurven für die Algorithmen *PIDA** und *AIDA** auf dem Torus

Kapitel 8

Zusammenfassung und Ausblick

In dieser Diplomarbeit wurde mit dem Algorithmus *AIDA** ein generischer Parallelisierungsansatz für Baumsuchverfahren vorgestellt. Ein großer Vorteil dieses Ansatzes ist, daß die initial erhaltene Aufteilung des Suchbaumes bestehen bleibt, so daß zu den einzelnen Paketen Informationen gesammelt werden können, welche in der Implementierung zu einer selbstausgleichenden dynamischen Lastverteilung führt. Weiter werden die Bäume in der dritten Phase durch die sequentielle Suchroutine expandiert, was zu einer hohen Arbeitsrate des parallelen Algorithmus führt. Außerdem können dadurch sequentielle Suchverfahren einfach parallelisiert werden.

Es wurde die gute Skalierbarkeit des Lastverteilungsverfahrens von *AIDA** sowohl anhand von empirischen Ergebnissen, als auch theoretisch nachgewiesen. Die in Kapitel 3 vorgestellte Isoeffizienz-Funktion liefert eine zuverlässige Aussage über die Skalierbarkeit von parallelen Systemen und ermöglicht den Vergleich unterschiedlicher paralleler Algorithmen auf verschiedenen parallelen Architekturen. Im Gegensatz zur Leistungsanalyse von sequentiellen Algorithmen, wo Speicherplatzbedarf und Rechenzeit abhängig von der Problemgröße den Aufwand eines Verfahrens beschreiben, kommen bei parallelen Systemen noch die Prozessorzahl und die Hardware-Architektur als weitere Parameter hinzu. Nicht jedes Verfahren ist auch für massiv parallele Systeme skalierbar, auch wenn es für kleinere Systeme gute Leistungen erreicht. So wurde etwa in dem letzten Graphen (Abbildung 7.2), welcher die Isoeffizienz-Funktionen für die Algorithmen *PIDA** und *AIDA** zeigt, deutlich, daß das in *PIDA** verwendete Lastverteilungsverfahren zur Auslastung von Systemen mit wenigen Prozessoren nur kleinere Probleme benötigt. Im Algorithmus *AIDA** muß dagegen die sequentielle erste Phase durch eine ausreichende Problemgröße ausgeglichen werden. Auf massiv parallelen Systemen ist dieses Verfahren mit der festen Aufteilung des Suchbaumes in der Initialphase dem anderen Ansatz deutlich überlegen, da die Initialverteilung im Algorithmus *PIDA** sehr aufwendig ist und hierdurch keine so gleichmäßige Aufteilung des Suchbaumes zu erreichen ist.

Die allgemeine Anwendbarkeit der Lastverteilungsstrategie mit einer initialen Aufteilung des

Suchbaumes wurde auch bei der Lösung der Floorplan-Optimierung deutlich. Der implementierte Branch&Bound–Tiefensuchalgorithmus zeigt, daß die erste Phase eine sehr ausgewogene Initialverteilung liefert, welche die Prozessoren am Anfang der Suche zu über 90% der Laufzeit auslastet, ohne daß eine dynamische Lastverteilung notwendig wird.

Das vorgestellte Lastverteilungsverfahren kann auf beliebige Baumsuchverfahren angewendet werden, wenngleich der Effekt der „lernenden“ Lastverteilung nur bei iterativen Verfahren auftreten kann. So kann man dieses Verfahren zur Parallelisierung von Spielbaumsuchen, Suche in UND/ODER-Bäumen oder anderen Branch&Bound-Methoden benutzen. Auch wäre es zur Anfangsverteilung bei einer parallelen Implementierung des Algorithmus A^* geeignet. Eine sinnvolle Anwendung dieses Algorithmus könnte auch die in dieser Arbeit vorgestellte Floorplan-Optimierung sein, da der Gesamtspeicher in einem massiv parallelen System ausreichend groß sein müßte. Allerdings ist bei der Implementierung von heuristischen Suchverfahren die Effizienz am sinnvollsten zu steigern, indem man die verwendete Heuristik verbessert, wie an der Implementierung der Floorplan-Optimierung in dieser Arbeit deutlich wurde.

Anhang A: Die Testreihen

Nr.	Ausgangsstellung	$h(r)$	$h^*(r)$	*sol*	Knotenzahl
1	14 1 9 6 4 8 12 5 7 2 3 0 10 11 13 15	35	45	1	668556
2	4 5 7 2 9 14 12 13 0 3 6 11 8 1 15 10	30	42	1	1060844
3	13 8 14 3 9 1 0 7 15 5 4 10 12 2 6 11	29	41	2	1100009
4	0 1 9 7 11 13 5 3 14 12 4 2 8 6 10 15	28	42	1	1146218
5	5 7 11 8 0 14 9 13 10 12 3 15 6 1 4 2	45	53	5	1477430
6	8 11 4 6 7 3 10 9 2 12 15 13 0 1 5 14	39	49	2	3342714
7	4 7 13 10 1 2 9 6 12 8 14 5 3 0 11 15	32	44	2	3836469
8	9 14 5 7 8 15 1 2 10 4 13 6 12 0 11 3	32	44	2	4645241
9	7 11 8 3 14 0 6 15 1 4 13 9 5 12 2 10	36	46	2	5069131
10	6 10 1 14 15 8 3 5 13 0 2 7 4 9 11 12	35	47	2	5150875
11	12 8 15 13 1 0 5 4 6 3 2 11 9 7 14 10	38	50	3	5369457
12	12 15 2 6 1 14 4 8 5 3 7 0 10 13 9 11	35	47	4	5841613
13	6 14 10 5 15 8 7 1 3 4 2 0 12 9 11 13	37	49	6	6184584
14	6 0 5 10 11 12 9 2 1 7 4 3 14 8 13 15	35	45	20	6708963
15	3 14 9 11 5 4 8 2 13 12 6 7 10 1 15 0	32	46	6	7414483
16	13 9 14 6 12 8 1 2 3 4 0 7 5 10 11 15	34	46	1	8620335
17	14 13 4 11 15 8 6 9 0 7 3 1 2 10 12 5	46	56	1	9137196
18	6 13 3 2 11 9 5 10 1 7 12 14 8 4 0 15	31	45	11	15860061
19	3 14 9 7 12 15 0 4 1 8 5 6 11 10 2 13	39	51	3	16120909
20	5 11 6 9 4 13 12 0 8 2 15 10 1 7 3 14	36	50	3	16617998
21	5 7 0 11 12 1 9 10 15 6 2 3 8 4 13 14	30	44	4	17511854
22	5 0 15 8 4 6 1 14 10 11 3 9 7 12 2 13	37	51	12	18839659
23	3 14 13 6 4 15 8 9 5 12 10 0 2 7 1 11	41	53	6	19989713
24	1 3 2 5 10 9 15 6 8 14 13 11 12 4 7 0	24	42	1	21063470
25	4 3 6 13 7 15 9 0 10 5 8 11 2 12 1 14	34	50	2	22647401

Tabelle A.1: Die Instanzen der Klasse I des 15–Puzzles (*all-solutions-case*, *sol* bezeichnet die Anzahl aller optimalen Lösungen)

Nr.	Ausgangsstellung	$h(r)$	$h^*(r)$	*sol*	Knotenzahl
26	3 6 5 2 10 0 15 14 1 4 13 12 9 8 11 7	36	46	1	24072829
27	13 0 9 12 11 6 3 5 15 8 1 10 4 14 2 7	39	53	2	24883099
28	7 15 8 2 13 6 3 12 11 0 4 10 9 5 1 14	41	53	6	28407490
29	5 4 7 1 11 12 14 15 10 13 8 6 2 0 9 3	36	50	5	31777446
30	13 14 6 12 4 5 1 0 9 3 10 2 15 11 8 7	36	52	31	32327749
31	14 7 1 9 12 3 6 15 8 11 2 5 10 0 4 13	36	52	2	36259150
32	7 8 3 2 10 12 4 6 11 13 5 15 0 1 9 14	31	47	6	39470149
33	6 0 14 12 1 15 9 10 11 4 7 2 8 3 5 13	43	55	1	41941365
34	10 9 3 11 0 13 2 14 5 6 4 7 8 15 1 12	33	49	3	51100896
35	8 4 6 1 14 12 2 15 13 10 9 5 3 7 0 11	35	49	32	54471011
36	1 7 15 14 2 6 4 9 12 11 13 3 0 8 5 10	35	49	7	56189088
37	9 0 4 10 1 14 15 3 12 6 5 7 11 13 8 2	35	49	11	56282925
38	12 5 13 11 2 10 0 9 7 8 4 3 14 6 15 1	39	53	1	65724501
39	6 12 11 3 13 7 9 15 2 14 8 10 4 1 5 0	36	52	3	73579735
40	4 7 14 13 10 3 9 12 11 5 6 15 1 2 8 0	42	56	20	74742146
41	6 11 7 8 13 2 5 4 1 10 3 9 14 0 12 15	36	52	17	78146885
42	13 5 4 10 9 12 8 14 2 3 7 1 0 15 11 6	43	55	17	81958256
43	4 6 12 0 14 2 9 13 11 8 3 15 7 10 1 5	43	57	12	84387585
44	0 13 2 4 12 14 6 9 15 1 10 3 11 5 8 7	34	54	8	89864844
45	12 3 9 1 4 5 10 2 6 11 15 0 14 7 13 8	31	49	3	91239607
46	12 11 15 3 8 0 4 2 6 13 9 5 14 1 10 7	32	50	2	94875202
47	12 6 0 4 7 3 15 1 13 9 8 11 2 14 5 10	36	52	9	104651083
48	5 9 13 14 6 3 7 12 10 8 4 0 15 2 11 1	43	57	1	154445918
49	12 9 0 6 8 3 5 14 2 4 11 7 10 1 15 13	32	50	15	161653537
50	10 2 8 4 15 0 1 14 11 13 3 6 9 7 5 12	44	56	27	168767116

Tabelle A.2: Die Instanzen der Klasse II des 15–Puzzles *(all-solutions-case)*

Nr.	Ausgangsstellung	$h(r)$	$h^*(r)$	*sol*	Knotenzahl
51	11 0 15 8 13 12 3 5 10 1 4 6 14 9 7 2	43	57	5	173085145
52	1 6 12 14 3 2 15 8 4 5 13 9 0 7 11 10	39	55	4	199963536
53	14 4 0 10 6 5 1 3 9 2 13 15 12 7 8 11	30	48	23	206926456
54	11 4 0 8 6 10 5 13 12 7 14 3 1 2 9 15	38	54	7	214762401
55	7 3 14 13 4 1 10 8 5 12 9 11 2 15 6 0	34	54	21	215562319
56	7 15 4 0 10 9 2 5 12 11 13 6 1 3 14 8	39	57	2	227574722
57	9 8 0 2 15 1 4 14 3 10 7 5 11 13 6 12	38	54	19	232936764
58	6 0 5 15 1 14 4 9 2 13 8 10 11 12 7 3	37	53	2	237629760
59	11 1 7 4 10 13 3 8 9 14 0 15 6 5 2 12	38	54	16	257638519
60	12 8 14 6 11 4 7 0 5 1 10 15 3 13 9 2	34	54	1	260254085
61	5 12 10 7 15 11 14 0 8 2 1 13 3 4 9 6	42	56	5	275641439
62	7 3 1 13 12 10 5 2 8 0 6 11 14 15 4 9	31	51	14	280066325
63	8 13 10 9 11 3 15 6 0 1 2 14 12 5 4 7	36	54	18	280376294
64	9 5 11 10 13 0 2 1 8 6 14 12 4 7 3 15	34	52	42	288137271
65	5 2 14 0 7 8 6 3 11 12 13 15 4 10 9 1	31	51	4	339429975
66	11 15 14 13 1 9 10 4 3 6 2 12 7 5 8 0	48	64	77	347157246
67	11 5 1 14 4 12 10 0 2 7 13 3 9 15 6 8	36	54	17	353979025
68	11 4 2 7 1 0 10 15 6 9 14 8 3 13 5 12	32	52	51	404388658
69	2 11 15 5 13 4 6 7 12 8 10 1 9 3 14 0	30	52	4	443303309
70	12 11 0 8 10 2 13 15 5 4 7 3 6 9 14 1	40	56	5	458908513
71	15 10 8 3 0 6 9 5 1 14 13 11 7 2 12 4	41	57	21	497817709
72	7 1 2 4 8 3 6 11 10 15 0 5 14 12 13 9	28	50	8	558341207
73	15 1 3 12 4 0 6 5 2 8 14 9 13 10 7 11	30	52	53	612105686
74	0 11 3 12 5 2 1 9 8 10 14 15 7 4 13 6	34	54	15	614187164
75	15 8 10 7 0 12 14 1 5 9 6 3 13 11 4 2	37	55	3	622680975

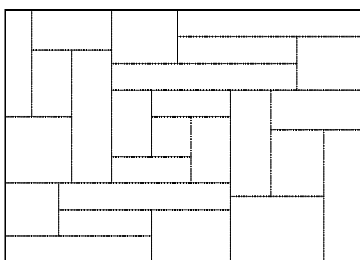
Tabelle A.3: Die Instanzen der Klasse III des 15–Puzzles *(all-solutions-case)*

Nr.	Ausgangsstellung	$h(r)$	$h^*(r)$	*sol*	Knotenzahl
76	14 13 15 7 11 12 9 5 6 0 2 1 4 8 10 3	41	57	1	685253879
77	10 8 0 12 3 7 6 2 1 14 4 11 15 13 9 5	38	56	10	843097128
78	14 1 8 15 2 6 0 3 9 12 10 13 4 7 5 11	33	53	31	867035987
79	14 3 9 1 15 8 4 5 11 7 10 13 0 2 12 6	41	59	2	929918801
80	14 10 9 4 13 6 5 8 2 12 7 0 1 3 11 15	43	59	23	970911213
81	14 3 5 15 11 6 13 9 0 10 2 12 4 1 7 8	42	60	18	1100622125
82	13 11 8 9 0 15 7 10 4 3 6 14 5 12 2 1	43	59	54	1110815047
83	9 7 5 2 14 15 12 10 11 3 6 1 8 13 0 4	41	57	5	1244938779
84	5 7 3 12 15 13 14 8 0 10 9 6 1 4 2 11	40	58	153	1302035868
85	13 11 4 12 1 8 9 15 6 5 14 2 7 3 10 0	44	62	26	1456639284
86	8 1 7 12 11 0 10 5 9 15 6 13 14 2 3 4	40	58	44	1654992137
87	7 6 8 1 11 5 14 10 3 4 9 13 15 2 0 12	41	59	1	1767759456
88	8 10 9 11 14 1 7 15 13 4 0 12 6 2 5 3	40	56	16	2195428891
89	14 7 8 2 13 11 10 4 9 12 5 0 3 6 1 15	41	59	91	2364629063
90	15 14 0 4 11 1 6 13 7 5 8 9 3 2 10 12	46	66	9	2506250734
91	14 9 12 13 15 4 8 10 0 2 1 7 3 11 5 6	50	64	21	3074438822
92	3 15 2 5 11 6 4 7 12 9 1 0 13 14 10 8	29	55	56	4904167160
93	3 2 7 9 0 15 12 4 6 11 5 14 8 13 10 1	37	57	126	5126176691
94	12 15 11 10 4 5 14 0 13 7 1 2 9 8 3 6	38	56	67	5221762876
95	15 14 6 7 10 1 0 11 12 8 4 9 2 5 13 3	35	57	39	5666440462
96	11 6 14 12 3 5 1 15 8 0 10 13 9 7 4 2	41	61	47	6906973331
97	10 0 2 4 5 1 6 12 11 13 9 7 15 3 14 8	33	59	19	7166866637
98	11 14 13 1 2 3 12 4 15 7 9 5 10 6 8 0	48	66	33	15226005098
99	14 10 2 1 13 9 8 11 7 3 6 12 15 5 4 0	40	62	2	19075654287
100	15 2 12 11 14 13 9 5 1 3 8 7 0 10 6 4	43	65	67	38152667568

Tabelle A.4: Die Instanzen der Klasse IV des 15–Puzzles *(all-solutions-case)*

Nr.	Ausgangsstellung	$h(r)$	$h^*(r)$	w_{seq}
1	9 6 2 0 18 17 5 1 7 4 11 3 16 14 13 15 12 19 8 10	39	63	10491786978
2	11 5 10 0 12 7 3 13 4 16 6 19 1 14 9 15 17 8 2 18	41	63	1502057584
3	8 17 7 2 16 0 9 3 5 13 15 18 4 6 19 12 10 1 14 11	44	64	157603091
4	10 9 1 2 8 16 3 14 13 12 19 15 6 17 5 18 4 0 11 7	45	65	38272200965
5	11 8 16 0 6 2 10 3 4 13 7 17 12 18 1 14 15 9 5 19	45	69	124309615395
6	12 11 9 0 13 5 14 4 10 1 6 19 16 17 15 2 7 8 18 3	45	69	3943413082
7	4 9 17 18 5 11 6 3 7 10 16 8 0 2 14 15 12 19 1 13	45	71	2372918004
8	16 7 6 5 17 13 1 19 15 8 11 4 12 0 2 18 9 14 10 3	48	72	36324106525
9	4 10 9 3 19 15 18 8 12 2 1 11 13 0 17 16 5 14 6 7	50	70	11481739248
10	9 2 13 4 6 0 3 17 16 12 10 11 15 7 14 18 5 19 8 1	50	74	56169269431
11	7 5 18 8 9 10 13 16 4 0 2 3 19 1 15 11 17 14 6 12	51	71	28987781870
12	6 13 8 5 11 0 10 14 16 17 1 7 19 9 15 3 2 4 18 12	52	74	80763103860
13	15 10 8 17 0 2 14 3 9 16 11 18 1 12 7 6 4 19 5 13	55	75	101945983497

Tabelle A.5: Die 13 durch *AIDA** gelösten Instanzen des 19-Puzzles



Problem	Kombinationen					
EX1	4×1	2×2	1×4			
EX2	6×1	3×2	2×3	1×6		
EX3	16×1	8×2	4×4	2×8	1×16	
EX4	12×1	6×2	4×3	3×4	2×6	1×12

Abbildung A.1: Der Floorplan und die Größenkombinationen der Blöcke für die vier Instanzen der Floorplan-Optimierung

Literaturverzeichnis

[Amdahl67] G. M. Amdahl, *Validity of the single-processor approach to achieving large scale computing capabilities*, AFIPS Conference Proceedings, vol. 30 (Atlantic City, N. J. April 18-20) AFIPS Press, Reston, Va. 1967, 483–485

[Annartone92] M. Annartone, *MPPs, Amdhal's Law and Comparing Computers*, Proceedings Frontiers on Parallel Computing 1992, 465–470

[BakerSchwarz83] B. S. Baker, J. S. Schwarz, *Shelf algorithms for two-dimensional packing problems*, SIAM Journal of Computing 12:3 (August 83), 189–302

[BauerKrieter88] B. Bauer, H. Krieter, *Parallelisierung problemlösender Methoden am Beispiel des $n \times n$-Puzzles*, Diplomarbeit, Fachbereich 17, Universität-GH Paderborn (1988)

[DetcherPearl88] R. Detcher, J. Pearl, *The Optimality of A**, in: Kanal, Kumar: Search in Artificial Intelligence, Springer Verlag (1988), 166–199

[DijkstraFeijenGastern83] E. Dijkstra, W. H. J. Feijen, A. J. M. van Gastern, *Derivation of a termination detection algorithm for distributed computation*, Information Processing Letters 16 (1983), 217–219

[FarrageMarsland93] S. Farrage, T. A. Marsland, *Dynamic Splitting of Decision Trees*, Technical Report TR93.03, Computing Science Department, University of Alberta, Edmonton (April 1993)

[FlattKennedy89] H. P. Flatt, K. Kennedy, *Performance of parallel processors*, Parallel Computing 12 (1989), 1–20

[GuptaKumar92] A. Gupta, V. Kumar, *Performance Properties of Large Scale Parallel Systems*, Technical Report 92-32, Department of Computer Science, University of Minnesota

[Gustafson88] J. L. Gustafson, *Reevaluating Amdahls Law*, Communications of the ACM, Volume 31, Number 5 (May 88), 532–533

[HartNilssonRaphael68] P. E. Hart, N. J. Nilsson, B. Raphael, *A formal basis for the heuristic determination of minimum cost paths*, IEEE Transactions on SSC 4 (1968), 100–107

[HartNilssonRaphael72] P. E. Hart, N. J. Nilsson, B. Raphael, *Correctionto 'A formal basis for the heuristic determination of minimum cost paths'*, SIGART Newsletter 37 (1972), 28–29

[Kaindl89] H. Kaindl, *Problemlösen durch heuristische Suche in der Artificial Intelligence*, Springer-Verlag (1989)

[Korf85] R. E. Korf, *Depth-first iterative-deepening: An optimal admissible tree search*, Artificial Intelligence 27 (1985), 97–109

[Korf88] R. E. Korf, *Optimal path-finding algorithms*, in: L. Kanal, V. Kumar (eds.), Search in Artificial Intelligence, Springer-Verlag (1988), 223–267

[KorfPowleyFerguson90] R. E. Korf, C. Powley, C. Ferguson, *Parallel Heuristic Search: Two Approaches*, in: Kumar, Gopalakrishnan, Kanal (eds.), Parallel Algorithms for Machine Intelligence and Vision, Springer-Verlag (1990), 42–65

[Korf93] R. E. Korf, *Linear-space-best-first search*, Artificial Intelligence 62 (1993), 41–78

[KruskalRudolphSnir90] C. P. Kruskal, L. Rudolph, M. Snir, *A complexity theory of efficient parallel algorithms*, Theoretical Computer Science 71 (1990), 95–132

[KumarRao90] V. Kumar, V. N. Rao, *Scalable parallel formulations of depth-first search*, in: Kumar, Gopalakrishnan, Kanal (eds.), Parallel Algorithms for Machine Intelligence and Vision, Springer-Verlag (1990), 1–41

[KumarSingh91] V. Kumar, V. Singh, *Scalability of parallel algorithms for the all-pairs shortest-path Problem*, Journal of Parallel and Distributed Computing 13 (1991), 124–138

[Lengauer90] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, J. Wiley & Sons – B. G. Teubner (1990)

[MahantiDaniels93] A. Mahanti, C. J. Daniels, *A SIMD approach to parallel heuristic search*, Artificial Intelligence 60 (1993), 243–282

[MorabitoArenalesArcaro92] R. N. Morabito, M. N. Arenales, V. F. Arcaro, *An and-or-graph approach for two-dimensional cutting-problems*, European Journal of Operational Research 58 (1992), 263–271

[Nilsson80] N. J. Nilsson, *Principles of Artificial Intelligence*, Springer-Verlag 1980

[PowleyFergusonKorf93] R. E. Korf, C. Powley, C. Ferguson, *Depth-first heuristic search on a SIMD machine*, Artificial Intelligence 60 (1993), 199–242

[RaoKumarRamesh87] V. N. Rao, V. Kumar, Ramesh, *A parallel implementation of iterative-deepening-a**, Procs. Nat. Conf. on AI (AAAI-87), 878–882

[RaoKumar93] V. N. Rao, V. Kumar, *On the Efficiency of Parallel Backtracking*, IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 4 (April 1993)

[Reinefeld92] A. Reinefeld, *Iterative Suche auf einem Transputer-Netzwerk*, TAT, Aachen (1992)

[ReinefeldMarsland93] A. Reinefeld, T. A. Marsland, *Enhanced iterative-deepening search*, Technical Report 120, Fachbereich 17, Universität-GH Paderborn (März 1993)

[Reinefeld93] A. Reinefeld, *Complete Solution of the Eight-Puzzle and the benefit of node-ordering in IDA**, Procs. Int. Joint Conf. on AI, Chambery, Savoi, France (Sept. 1993)

[ReinefeldSchnecke94] A. Reinefeld, V. Schnecke, *AIDA* – Asynchronous Parallel IDA**, 10th Canadian Conf. on Artificial Intelligence AI '94, Banff, Canada (May 1994)

[ReinefeldSchnecke94a] A. Reinefeld, V. Schnecke, *Work-Load Balancing in Highly Parallel Depth-First Search*, IEEE-SHPCC '94, Knoxville, Tennesse (May 1994)

[SarkarChakrabartiGhose92] U. K. Sarkar, P. P. Chakrabarti, S. Ghose, S. C. De Sarkar, *Effective use of memory in iterative deepening search*, Information Processing Letters 42 (1992), 47–52

[Snir92] M. Snir, *Scalable Parallel Computers and Scalable Parallel Codes: From Theory to Practice*, 1st Heinz Nixdorf Symposium on Parallel Architectures and their Efficient Use, November 1992

[WimerKorenCederbaum88] S. Wimer, I. Koren, I. Cederbaum, *Optimal aspect ratios of building blocks in VLSI*, 25th ACM/IEEE Design Automation Conference (1988), 66–72

Dank

Ich danke Dr. Alexander Reinefeld für die interessante Themenstellung und intensive Betreuung während der gesamten Bearbeitungszeit. Weiter danke ich Prof. Dr. Franz-J. Rammig für die Übernahme des Zweitgutachtens.

Außerdem möchte ich mich recht herzlich bei allen Mitarbeitern des PC^2 für die freundliche Hilfe während der Implementierungsphase bedanken.

Mein besonderer Dank gilt Bernard Bauer für seine engagierte Unterstützung und viele Anregungen bei der Erstellung der Ausarbeitung.

Versicherung

Ich versichere, daß ich diese Diplomarbeit ohne die Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.