Parallele Datenverarbeitung aktuell: TAT'94, IOS Press, 1994, 178-187

Iterative Tiefensuche auf einem massiv parallelen System

Volker Schnecke
Universität Osnabrück
Fachbereich Mathematik/Informatik
49069 Osnabrück
volker@informatik.uni-osnabrueck.de

Zusammenfassung. In diesem Artikel wird ein allgemeiner Parallelisierungsansatz für Baumsuchalgorithmen vorgestellt. Der Kern dieses Ansatzes liegt in einer zweiphasigen Lastverteilungsstrategie, durch die es möglich ist, ohne jede Kommunikation oder Synchronisation eine initiale Aufteilung des Suchbaumes zu erzeugen, die so ausgewogen ist, daß in weniger als 10% der Gesamtsuchzeit eine dynamische Lastverteilung erfolgt. Als Anwendung wurde ein einfaches Spielproblem, das $n \times n$–Puzzle gewählt. Die Implementierung erfolgte unter PARIX und wurde sowohl auf dem Parsytec GCel als auch auf einem Netz von Parsytec Power Xplorern getestet. Dabei werden selbst für sehr kleine Instanzen mit einer parallelen Laufzeit von weit unter einer Minute Speedups von über 700 auf einem System mit 1024 Prozessoren erreicht.

1. Einleitung

Suche ist neben *Wissen* einer der Stützpfeiler der künstlichen Intelligenz. Während der Suche wird ein Entscheidungsbaum durchlaufen, der durch ein Produktionssystem bzw. ein gegebenes Optimierungsproblem beschrieben wird. Im Rahmen eines Optimierungsproblems ist der günstigste (z. B. kürzeste) Weg von der Wurzel des Baumes, welche den Ausgangszustand des Systems beschreibt, zu einem Lösungsknoten (Zielzustand) gesucht [8].

Im folgenden werden zunächst ein spezielles Suchverfahren, die *iterative Tiefensuche* und eine Anwendung vorgestellt. Nach einer kurzen Beschreibung von alternativen Parallelisierungsansätzen wird der parallele iterative Tiefensuchalgorithmus *AIDA** vorgestellt. Es werden die Implementierungen von AIDA* unter PARIX und PVM beschrieben und die Ergebnisse für die PARIX-Version aufgeführt.

2. Iterative Tiefensuche

Die *iterative Tiefensuche* ist ein Baumsuchverfahren, welches die Vorteile von Tiefen– und Breitensuche in sich vereint. Bei der *Breitensuche* wird der Suchbaum Ebene für Ebene durchsucht. Dieses hat den Vorteil, daß der erste gefundene Lösungsknoten auch ein optimaler Lösungsknoten ist, da alle höheren Ebenen des Baumes zuvor ohne Erfolg durchsucht worden sind. Der große Nachteil der Breitensuche liegt jedoch in dem Speicherplatzbedarf, da bei diesem Verfahren alle Knoten einer Ebene des Suchbaumes im Speicher gehalten werden müssen. Die Knotenzahl einer Ebene des Baumes wächst in der Regel exponentiell mit der Suchtiefe.

Bei der *Tiefensuche* wächst der Speicherplatzbedarf nur linear mit der Suchtiefe. Hier kann jedoch die Suche beliebig in die tieferen Ebenen des Baumes laufen, so daß dabei unnötig viele Knoten betrachtet werden. Dieses kann durch die Einführung einer Tiefenschranke verhindert werden. Diese Schranke sorgt dafür, daß die Suche an einer Stelle abgebrochen wird, wenn eine bestimmte Tiefe erreicht ist. Die Schwierigkeit liegt dabei jedoch in der Wahl dieser Schranke: Ist sie zu niedrig, so wird kein Lösungsknoten expandiert, wird sie zu groß gewählt, so werden unnötig viele Knoten betrachtet. In der iterativen Tiefensuche werden eine Reihe von Tiefensuchen mit wachsender Tiefenschranke durchgeführt, wobei die Schranke nach einer erfolglosen Iteration um einen minimalen Wert erhöht wird. Dadurch wird sichergestellt, daß die erste gefundene Lösung auch optimal ist.

Zur Steigerung der Effizienz bei der Suche ist es sinnvoll, durch problemspezifische Informationen die Suche auf die „günstigeren" Bereiche des Baumes zu beschränken. Diese *heuristischen* Informationen kommen in einer Knotenbewertungsfunktion zum Ausdruck und geben die geschätzten Kosten für einen Pfad von einem Knoten im Suchbaum zu einem Lösungsknoten an. Eine typische Knotenbewertungsfunktion, wie sie auch in dem Bestensuch–Algorithmus A^* [8] verwendet wird, setzt sich für einen Knoten n wie folgt zusammen:

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

Dabei bezeichnet $h(n)$ die heuristische Funktion und $g(n)$ beschreibt die minimalen Kosten für einen Pfad von der Wurzel des Suchbaumes zu diesem Knoten n. Die Knotenbewertungsfunktion $f(n)$ repräsentiert somit die voraussichtlichen Kosten für den günstigsten Pfad von der Wurzel zu einem Lösungsknoten, welcher durch den Knoten n verläuft.

Der heuristische iterative Tiefensuchalgorithmus *IDA* (Iterative deepening A*)* [4] verwendet die beschriebene Bewertungsfunktion, um *kostenbeschränkte* Tiefensuchen durchzuführen. Im Gegensatz zur reinen iterativen Tiefensuche ist nicht die Tiefe eines Knotens entscheidend dafür, ob dieser Knoten in der aktuellen Iteration noch weiter expandiert wird, sondern der Wert $f(n)$ der Bewertungsfunktion. Wird die Kostenschranke zwischen den Iterationen um einen minimalen Wert erhöht und überschätzt die heuristiche Funktion die tatsächlichen Kosten nicht, so ist sichergestellt, daß der Algorithmus IDA* einen optimalen Lösungsknoten, d. h. einen Knoten mit minimalen Kosten, findet, falls dieser existiert. Wächst die Zahl der in einer Iteration betrachteten Knoten exponentiell mit der Iterationszahl, so ist der durch die redundanten Knotenexpansionen in den einzelnen Iterationen entstandene Mehraufwand vernachlässigbar, da der Algorithmus *IDA** in diesem Fall asymptotisch den gleichen Zeitaufwand hat wie der optimale Algorithmus A* [4].

3. Anwendungen der iterativen Tiefensuche

Die iterative Tiefensuche findet Anwendungen in der *Operations Research* und in der *Künstlichen Intelligenz*. Dieses Verfahren kann etwa für zweidimensionale Pack–Probleme *(Cutting-Stock Problem)* [14] oder im Bereich des VLSI-Chip-Entwurfs *(Floorplan–Area–Optimization)* [15] angewendet werden.

Typische Anwendungen der iterativen Tiefensuche zeichnen sich durch die folgenden Eigenschaften aus:

- geringe Lösungsdichte

- hoher Verzweigungsfaktor des Suchbaumes
- schlechte Abschätzbarkeit von oberen bzw. unteren Schranken für die Lösung

Für Probleme mit diesen Eigenschaften versagen etwa die gängigen *Branch & Bound*–Verfahren, da der durchsuchte Baum aufgrund der schlechten Schranken unnötig groß ist. Die praktische Anwendbarkeit des Bestensuch–Algorithmus A* auf diese Probleme ist ebenfalls sehr gering, da in diesem Verfahren – wie bei der Breitensuche – die gesamte Suchfront im Speicher gehalten werden muß, was in der Regel sehr bald zu einem Speicherüberlauf führt.

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

$\longrightarrow$

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

Abbildung 1: Das 4×4– bzw. 15–Puzzle in einer beliebigen Ausgangsstellung (links) und in der Zielstellung (rechts)

Eine einfache Anwendung mit den oben genannten Eigenschaften ist das $n \times n$ – *Puzzle* [8]. Das n×n –Puzzle besteht aus einem festen Rahmen der Größe n×n, in dem sich $n^2 - 1$ durchnumerierte Spielsteine befinden, ein Feld ist leer. In einem Zug kann nun ein zum leeren Feld benachbarter Spielstein auf dieses Feld verschoben werden. Die Lösung des n×n–Puzzles besteht darin, eine beliebige Ausgangsstellung mit einer minimalen Anzahl von Zügen in eine definierte Zielstellung zu überführen. Das n×n–Puzzle ist ein $\mathcal{NP}$–vollständiges Problem [9].

Der Algorithmus IDA* ist das bislang einzige Verfahren, welches kleinere Instanzen, wie etwa das 4×4–Puzzle mit annehmbaren Ressourcen lösen kann. Dabei werden bei 100 zufälligen Instanzen im Durchschnitt $363 \cdot 10^6$ Knoten betrachtet [4]. Als Heuristik wird die *Manhattan–Distanz* verwendet, welche sich aus der Summe der kürzesten Distanzen aller Steine von ihren Zielpositionen berechnet. Während der Fehler dieser Heuristik im Falle des 4×4–Puzzles noch relativ gering ausfällt, hat sich diese Heuristik zur Lösung des 4×5–Puzzles als nicht sehr geeignet erwiesen, da die sich ergebenden Suchbäume trotz der Verwendung einer Heuristik noch sehr groß sind [13]. So wurden bei der Lösung einer zufälligen Instanz dieses Problems mehr als 1.5 Billionen Knoten expandiert.

4. Parallelisierungsansätze

Baumsuchalgorithmen eignen sich aufgrund der feingranularen Parallelität sehr gut zur Parallelisierung – selbst auf massiv parallelen Systemen mit mehr als eintausend Prozessoren. Die Schwierigkeit dabei liegt jedoch in der Lastverteilung. Der Baum kann beliebig in Teilbäume zerlegt werden, welche unabhängig voneinander durchsucht werden können. Bei der Aufteilung des Suchbaumes unter den Prozessoren ist die Größe der sich dabei ergebenen Teilbäume jedoch nicht *a priori* bekannt. Aus diesem Grund ist eine effiziente dynamische Lastverteilungsstrategie notwendig, um die Prozessoren während der Suche möglichst gleichmäßig auszulasten.

Ein weiteres Problem stellt die initiale Lastverteilung in einem massiv parallelen System dar. Bei Verfahren, die auf einem *Stack–Splitting* Ansatz basieren [2, 10], beginnt ein Prozessor mit der Suche und baut dabei einen Stack mit unbearbeiteten Knoten auf. Falls dieser Stack eine bestimmte Tiefe erreicht hat, wird er geteilt, d. h. dieser Prozessor gibt einen oder mehrere Knoten an jeden seiner Nachbarn ab, die nach dem gleichen Verfahren wiederum Arbeit für ihre Nachbarprozessoren erzeugen. Dieses Verfahren setzt sich während der gesamten Suche fort. Ein Prozessor, der seinen erhaltenen Stack abgearbeitet hat, stellt wieder eine Arbeitsanfrage an einen seiner Nachbarn *(Task–Attraction Scheme)*.

In einem massiv parallelen System führt dieser Ansatz dazu, daß die Prozessoren mit einem größeren Abstand zum Startprozessor durch das rekursive Aufsplitten des Stacks nur sehr kleine Lastpakete erhalten und somit viele Anfragen stellen müssen [7].

Ein anderer Parallelisierungsansatz ist die *parallele Fenstersuche* [6]. Hier beginnen alle Prozessoren ihre Suche an der Wurzel des Suchbaumes, jedoch mit unterschiedlichen Kostenschranken. Dieses Verfahren ist für die Parallelisierung auf einem massiv parallelen System nicht geeignet, außerdem kann die Optimalität einer gefundenen Lösung nicht garantiert werden.

5. Der Algorithmus AIDA*

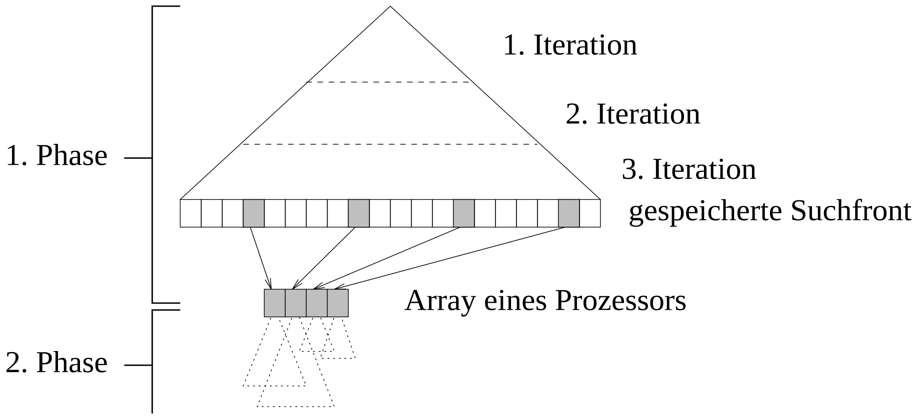

Abbildung 2: Die zwei Phasen des Algorithmus AIDA*

Der Algorithmus *AIDA* (Asynchronous parallel IDA*)* verwendet ein zweiphasiges Lastverteilungsverfahren, welches nicht auf einem Stack–Splitting, sondern auf einem *Search–Frontier–Splitting* (Aufteilen der Suchfront anstelle von Aufteilen des Stacks) basiert. Dabei wird der Baum bis zu einer bestimmten Tiefe expandiert, wobei die sich dabei ergebende Suchfront unter den Prozessoren des Systems aufgeteilt wird.

In der Initialverteilungsphase beginnen alle Prozessoren redundant damit, die ersten Iterationen auszuführen, wobei die Knoten der jeweiligen Suchfront in einem Array in jedem Prozessor gespeichert werden. Hat diese Suchfront eine bestimmte Größe erreicht, so wird sie gleichmäßig unter den Prozessoren aufgeteilt. Ein Prozessor P_i erhält dabei die Knoten $n_i, n_{i+p}, n_{i+2p}, \ldots$, so daß die Suchfront breit über das System verteilt wird. Dieses ist ein großer Vorteil gegenüber den Stack–Splitting Verfahren, wo sich die Suchfront durch das System „schiebt“, so daß im Baum benachbarte Knoten auch im System nah beieinander liegen. Ein weiterer Vorteil dieser Initialverteilungsstrategie ist, daß sie

– bis auf das Verbreiten der Wurzel – ohne jede Kommunikation oder Synchronisation erfolgt.

In der zweiten Phase werden – ausgehend von den Knoten der gespeicherten Suchfront – die Teilbäume unterhalb dieser Knoten durchsucht. Dabei bleibt die in der ersten Phase erzeugte Aufteilung der Suchfront über alle Iterationen erhalten, so daß in jeder Iteration die Suche bei diesen Knoten beginnt. Ein Prozessor, der alle „seine" Knoten abgearbeitet hat, stellt eine Anfrage an einen seiner Nachbarprozessoren. Falls dieser noch unbearbeitete Knoten in seinem Array hält, schickt er einige davon an seinen Nachbarn zurück, ansonsten leitet er die Anfrage an einen anderen Prozessor weiter. Knoten, die während dieser dynamischen Lastverteilung verschickt werden, wechseln ihren „Besitzer", d. h. sie werden aus dem Array des Versenders gestrichen und in das Array des Empfängers übernommen. Hierdurch reduziert sich eine evtl. durch die Initialverteilung entstandene ungleichmäßige Lastverteilung mit der Zahl der Iterationen [11]. Ein Prozessor, der in seiner Nachbarschaft keine unbearbeiteten Knoten gefunden hat, beginnt sofort mit der folgenden Iteration, so daß zwischen den Iterationen keine globale Synchronisation wie etwa in [10] erfolgt[1]. Hat ein Prozessor einen Lösungsknoten expandiert, so wird dieses den anderen Prozessoren mitgeteilt, die die Suche sofort abbrechen. Um die Optimalität dieser Lösung zu garantieren, muß jedoch beachtet werden, daß die vorherige Iteration komplett bearbeitet wurde.

Ein wesentlicher Vorteil des Algorithmus AIDA* liegt darin, daß auch für die parallele Suche die sequentielle Knotenexpansionsroutine verwendet wird. Dieses liefert eine maximale Arbeitsrate und ermöglicht eine einfache Parallelisierung von anderen sequentiellen Anwendungen. Bei dem Stack–Splitting Ansatz muß die rekursive Knotenexpansionsroutine angepaßt werden, da hier explizit ein Stack zur Verwaltung der Teilprobleme zur Lastverteilung implementiert werden muß.

6. Implementierung unter PARIX

Die Implementierung von AIDA* [13] erfolgte auf dem Parsytec GCel des PC^2 unter dem Betriebssystem PARIX. Als virtuelle Topologien wurden der Ring und der zweidimensionale Torus verwendet. Aus Effizienzgründen wurden nicht die Bibliotheksroutinen für die asynchrone Kommunikation verwendet, sondern die asynchrone Kommunikation durch zwei Threads für jeden Kanal implementiert.

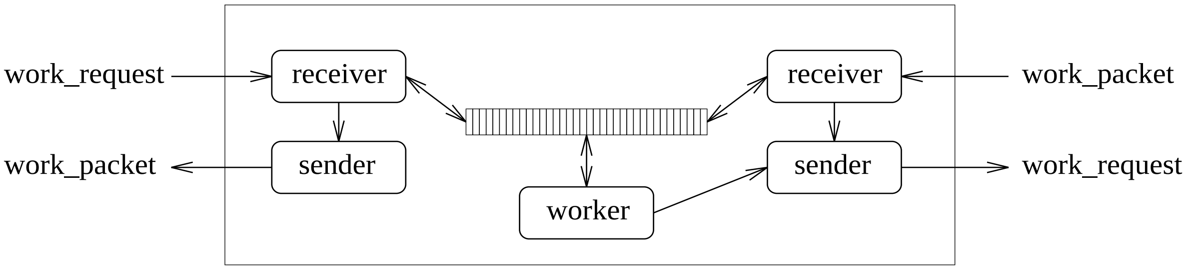

Abbildung 3: Das Prozeßmodell für die Implementierung von AIDA* unter PARIX auf der Ring-Topologie

Abbildung 3 zeigt das Prozeßmodell für die Implementierung auf der Ring–Topologie. Der `worker`– und die `receiver`–Threads greifen auf das Array mit den Knoten

[1] Im Falle des n×n–Puzzles ist das Inkrement der Kostenschranke zwischen den Iterationen immer 2. Ist für eine Anwendung dieser Wert nicht im voraus klar, so muß nach jeder Iteration das minimale Inkrement global ermittelt werden, was eine Synchronisation zwischen den Iterationen erfordert.

der gespeicherten Suchfront zu. Die Kommunikations-Threads (`sender` und `receiver`) bearbeiten Arbeitsanfragen von Nachbarprozessoren bzw. schicken eine Arbeitsanfrage des `worker` an einen Nachbarprozessor.

Nachdem alle Prozessoren in der ersten Phase eine Initialverteilung geschaffen haben, welche – bis auf das Verbreiten der Wurzel – ohne jede Kommunikation erfolgt, beginnen alle `worker` im System unabhängig voneinander mit der Suche in den Teilbäumen unter den Knoten der gespeicherten Suchfront. Eine erste Phase von wenigen Sekunden liefert eine Initialverteilung, welche die Prozessoren zu über 90% der Gesamtsuchzeit auslastet, ohne das dynamische Lastverteilung nötig wird. Zu Beginn der zweiten Phase hält jeder Prozessor ca. 1000 Knoten. Hat ein Prozessor alle Knoten aus seinem Array abgearbeitet, so läßt durch den `sender` zum rechten Nachbarn auf dem Ring eine Arbeitsanfrage verschicken. Die Anfrage bewegt sich auf dem Ring weiter, bis ein Prozessor Arbeit abgeben kann. Ein solches Arbeitspaket bewegt sich in entgegengesetzter Richtung zum anfragenden Prozessor zurück. Konnte kein Prozessor auf dem Ring Arbeit abgeben, so erreicht die Anfrage wieder ihren Absender, der sofort mit der nächsten Iteration beginnen kann, da im System keine freien Knoten mehr vorhanden sind. Es wird dabei bewußt in Kauf genommen, daß noch Arbeitspakete im Umlauf sein können. Um dieses festzustellen müßte ein Terminierungserkennungsalgorithmus [1] verwendet werden. Da die Paketgröße jedoch beschränkt ist, wird eine Überlappung der Iterationen unter den einzelnen Prozessoren des Systems toleriert.

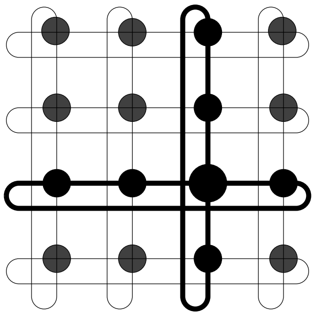

Abbildung 4: Die von einem Prozessor bei der Lastverteilung berücksichtigten Prozessoren auf dem vertikalen bzw. horizontalen Ring

Obwohl dieses Lastverteilungsverfahren auf der Ring–Topologie gute Ergebnisse liefert, so ist es doch für größere Systeme weniger geeignet, da die Nachrichten am Ende jeder Iteration sehr weite Wege zurücklegen. Aus diesem Grund wurde die Implementierung auf die Torus–Topologie übertragen. Im entsprechenden Prozeßmodell kommen für jeden der beiden zusätzlichen Kanäle `sender`– und `receiver`–Threads hinzu. Bei der Lastverteilung werden zunächst die Prozessoren auf dem horizontalen Ring nach dem oben beschriebenen Verfahren berücksichtigt. Erst wenn auf diesem Ring keine freien Knoten mehr vorhanden sind, wird die Anfrage auf den vertikalen Ring des Absenders weitergeleitet (siehe Abbildung 4). Dieses Lastverteilungsverfahren hat sich als sehr effizient erwiesen, da jeder Prozessor eine andere Menge von $2 \cdot \sqrt{p}$ Prozessoren des Systems berücksichtigt. Eine Überlast wird somit schnell breit über das System verteilt abgebaut und der Kommunikationsoverhead wächst nicht linear mit der Systemgröße [12].

7. Ergebnisse

Die Implementierung von AIDA* unter PARIX wurde auf dem Parsytec GCel-1024 des PC^2 und auf einem Netz von Parsytec Power Xplorern bei der ZIAM in Aachen getestet. Als Problem wurde das 4×4– bzw. 4×5–Puzzle gelöst. Im Falle des 4×4–Puzzles wurden die 100 Instanzen aus [4] in vier Klassen mit je 25 Instanzen eingeteilt.

	4×4–Puzzle		4×5–Puzzle
	Klasse III	Klasse IV	
Knoten	128,150,289	1,287,164,777	108,218,162,497
T_{seq}	0.9 h	10.2 h	30.6 d
$T(1024)$	7.1 sec	43.1 sec	29.5 min
$sp(1024)$	488.8	706.9	956.3
$eff(1024)$	47.7%	69.0%	93.5%

Tabelle 1: Die mittleren Werte für die Instanzen des Puzzle-Problems

Die Tabelle 1 zeigt die Mittelwerte für die einzelnen Problemklassen. Dabei werden für das 4×4–Puzzle nur die Klassen III und IV aufgeführt, da mit den 50 kleineren Instanzen ein System von mehr als eintausend Transputern nicht sinnvoll ausgelastet werden kann. Die durchschnittliche parallele Laufzeit auf 1024 Prozessoren liegt selbst in der Klasse IV nur bei 43 Sekunden. Es wird in dieser Klasse ein durchschnittlicher Speedup von 707 erreicht, wobei die Dauer für die erste Phase, in der alle Prozessoren redundant dieselben Knoten expandieren, etwa drei Sekunden beträgt. Für einzelne Probleme liegt der Speedup sogar über 840. Die angegebenen Speedup-Werte sind normiert über die Zahl der Knotenexpansionen in einer Sekunde und somit frei von Speedup–Anomalien[2]. Im Falle des 4×5–Puzzles wurden 13 kleinere zufällig erzeugte Instanzen getestet. Die Laufzeiten und Speedup-Werte für die einzelnen Instanzen sind in Tabelle 2 aufgeführt.

Die PARIX–Implementierung wurde auch auf einem Netz von 16 Parsytec Power Xplorern mit jeweils 4 Motorola PowerPC Prozessoren getestet. Die Leistung des PC 601 liegt bei der sequentiellen Implementierung des Algorithmus IDA* für das 4×4–Puzzle um einen Faktor von etwa 24 höher als die des T805. Leider ist die Kommunikation in diesem System deutlich teurer im Vergleich zur Rechenzeit als etwa im GCel. Hierdurch schrumpft dieser Faktor bei den 13 kleineren Testinstanzen von 24 (sequentielle Version) auf 6 (64 Prozessoren). Die mittlere Laufzeit liegt dabei bei etwa 6 Sekunden pro Instanz (GCel: 36 Sekunden). Die Abbildung 6 zeigt die durchschnittlichen Speedups für die 13 Instanzen auf den beiden Systemen.

8. Implementierung unter PVM

Bei der Implementierung von AIDA* unter PVM konnte das in Abbildung 3 dargestellte Prozeßmodell nicht beibehalten werden. Jeder Task in PVM ist ein Unix–Prozeß und verfügt somit über seinen eigenen Speicherbereich, so daß ein Array, auf das Worker– und Kommunikations–Tasks gemeinsam zugreifen, nicht möglich ist. Die Abbildung 7 zeigt das Prozeßmodell für die Implementierung unter PVM. Dabei wird das

[2] In Baumsuchverfahren betrachten der parallele und der sequentielle Algorithmus nicht die gleichen Knoten, was zu einem überlinearen Speedup führen kann. Bei der iterativen Tiefensuche ist dieser Effekt jedoch nicht sehr stark ausgeprägt, da die Knotenzahlen lediglich in der letzten Iteration abweichen können.

Nr.	$T(512)$	$sp(512)$	$T(768)$	$sp(768)$	$T(1024)$	$sp(1024)$
1	483	483.8	304	722.9	227	960.4
2	254	481.0	71	701.2	127	952.8
3	156	480.1	74	651.6	48	821.7
4	1565	488.6	1063	731.5	1036	975.7
5	19543	491.1	12114	721.4	9111	1005.2
6	238	479.7	106	709.2	87	942.5
7	1337	483.8	881	725.2	1119	963.2
8	1656	482.2	1696	721.4	984	958.8
9	609	484.4	783	718.2	398	945.2
10	2988	484.0	1906	725.4	1189	989.3
11	1636	482.5	1285	719.9	916	957.9
12	4691	483.6	2772	719.0	2177	989.0
13	11135	486.3	6221	724.6	5608	970.8
	561	483.9	2252	714.7	1771	956.3

Tabelle 2: Die 13 Instanzen des 4×5–Puzzles auf dem GCel

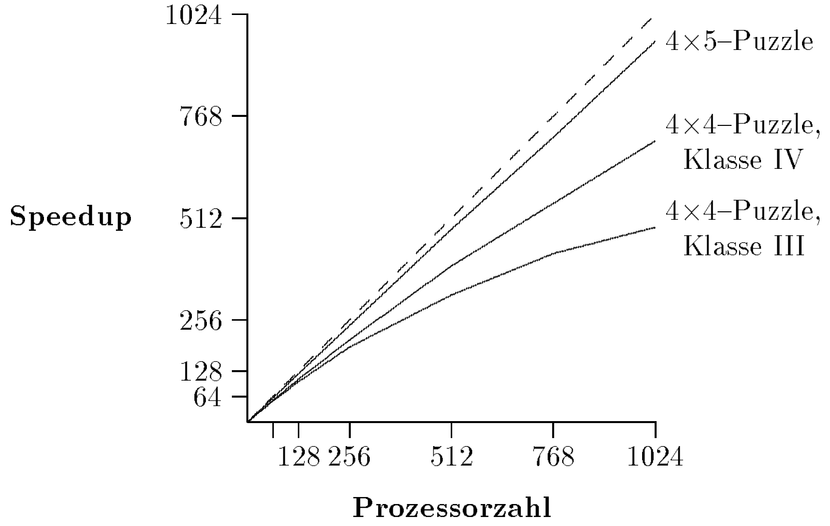

Abbildung 5: Die Speedupgraphen für das Puzzle-Problem auf dem Parsytec GCel-1024 des PC²

Array dem Kommunikations–Task zugeordnet, der auch die erste Phase ausführt. Dieser startet den Worker–Task auf demselben Host, und schickt ihm auf Anfrage Knoten aus dem Array. Hat ein Worker alle Knoten aus dem Array „seines“ Kommunikations–Tasks abgearbeitet, so verschickt dieser eine Arbeitsanfrage an einen Nachbar–Task. Obwohl alle Hosts über einen gemeinsamen Bus verbunden sind, werden die Anfragen unter den Prozessoren wie auf einem Ring der Reihe nach an alle Prozessoren weitergeleitet. Lediglich die Arbeitspakete werden direkt an den anfragenden Prozessor zurückgeschickt.

9. Zusammenfassung

Mit dem Algorithmus AIDA* wurde ein generischer Parallelisierungsansatz für Baumsuchalgorithmen vorgestellt. Die zweiphasige Lastverteilungsstrategie liefert ohne jede

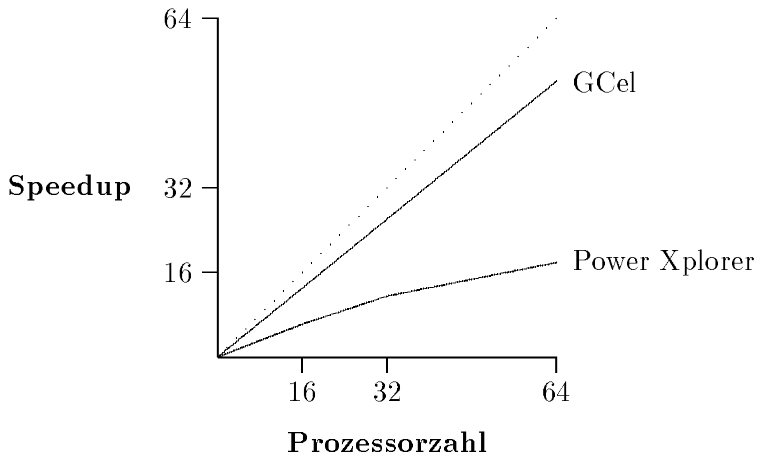

Abbildung 6: Die Speedupgraphen für AIDA* auf dem Xplorer und dem GCel

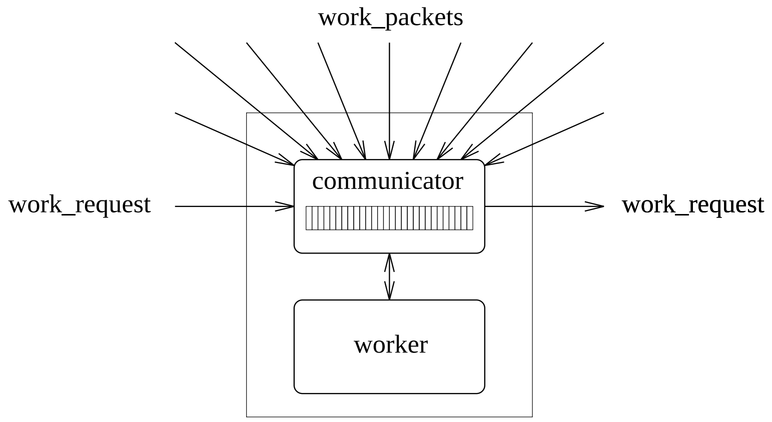

Abbildung 7: Das Prozeßmodell für die Implementierung von AIDA* unter PVM

Kommunikation und Synchronisation selbst auf einem massiv parallelen System eine ausgewogene Initialverteilung. Die weitere Suche erfolgt mit der optimalen sequentiellen Knotenexpansionsroutine und liefert somit eine höchstmögliche Leistung. Das verwendete Lastverteilungsverfahren zeichnet sich durch eine gute Skalierbarkeit aus und arbeitet selbst für kleine Instanzen auf Systemen mit über eintausend Prozessoren sehr effizient. Der Search–Frontier–Splitting Ansatz ist anderen Verfahren (z. B. Stack–Splitting) deutlich überlegen und wurde auch schon zur Parallelisierung eines nicht-iterativen Tiefensuchalgorithmus erfolgreich eingesetzt [12]. Die Portierung der Implementierung von PARIX nach PVM erforderte eine Änderung des Prozeßmodells, die Lastverteilungsstrategie wurde beibehalten. Der Parallelisierungsansatz liefert – in den ersten Testläufen – auch für ein Workstation–Cluster gute Ergebnisse. Hier kann jedoch noch weiter optimiert werden.

Dank

Besonderer Dank gilt Alexander Reinefeld für das interessante Thema und die sehr gute Betreuung der Diplomarbeit [13] am Paderborner PC^2, deren Ergebnisse diesem Artikel zugrundeliegen.

Literatur

[1] 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

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

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

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

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

[6] 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

[7] 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

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

[9] D. Ratner, M. Warmuth, *Finding a shortest solution for the $N \times N$ extension of the 15–puzzle is intractable*, AAAI-86, 168–172

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

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

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

[13] V. Schnecke, *Iterative Tiefensuche auf einem massiv parallelen System*, Diplomarbeit, Universität-GH Paderborn, PC^2 (März 94)

[14] C. Whitlock, N. Christofides, *An Algorithm for Two-Dimensional Cutting Problems*, Operations Research, Vol. 25 No. 1 (1977), 30–44

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