Overhead des Algorithmus ist gering und enthält hauptsächlich zusätzliche Programm-Counter. Überdies hat der CG-Algorithmus einge Anpassungsfähigkeiten, welche Graphen mit höherem Außengrad schneller verteilt.
Verbesserungen und Erweiterungen für den CG-Algorithmus, wären ein verfeinertes Auswahlkriterium, wann ein Task gespawnes werden soll, basierend auf Rekursionstiefen für verschiedene Funktionen während der Laufzeit.

8. Referenz

Gad Aharoni, Dror G. Feitelson and Amnon Barak, A run-time algorithm for managing the granularity of parallel functional programs. Departement of Computer Science, The Hebrew University of Jerusalem, Israel

Arvind and Nikhil, R.S. 1990. Executing a program on the MIT tagged-token dataflow architecture. IEEE Trans. Comput. 39 (3)

Debray, S.K., Lin, N.-W. and Hermenegildo, M. 1990. Task granularity analysis in logic programs. Programming Languages Design and Implementation, ACM SIGPLAN, White Plans, New York

Eager, D.L., Lazowska, E.D. and Zahorjan, J. 1986. Adaptive load sharing in homogeneous distributed systems. IEEE Trans. Softw. Eng. 12 (5)

Lin, F.C.H. and Keller, R.M. 1987. The gradient model load balancing method, IEEE Trans.Softw. Eng. 13 (1)

Configuration	Time [sec]	Speedup	Time [sec]	Speedup	Time [sec]	Speedup
Serial	3660.3	1	3062.0	1	126.5	1
1 PE	3935.0	0.93	3290.3	0.93	135.9	0.93
2 PEs	1980.3	1.9	1672.3	1.8	139.0	0.91
4 PEs	1024.5	3.6	877.2	3.5	143.6	0.88
8 PEs	541.1	6.8	478.7	6.4	142.4	0.89

Funktionen:

fib(0) = 1
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2)

$power_2(0) = 1$
$power_2(n) = power_2(n-1) + power_2(n-1)$

load(0) = 1
load(n) = load(n-1) and load(n-1)
comb'(0,n) = 1
comb'(h,n) = comb'(h-1) + load(n)

Def.: Speedup

Der Speedup wird durch das Verhältnis T_s / T_p bestimmt, wobei T_s die, für die serielle Ausführung der Funktion, benötigte Zeit und T_p die parallele Ausführungszeit ist. P bezeichnet die Anzahl der verwendeten PEs.

7. Fazit

Es wurde ein Online-Algorithmus beschrieben, der die Parallelität des Programms kontrolliert, um so eine effektive Granularität zu erreichen. Der vorgestellte CG-Algorithmus löst dieses Problem wesentlich besser, als herkömmliche Strategien. Weiterhin gibt es keinen konsequent besseren Algorithmus. Dieser Algorithmus erhöht die Granularität, indem er Parallelität ausnutzt wo sie existiert und ineffektive Parallelität ignoriert (bei zu kleinen Tasks). Der

Der parallele Task benötigt mehr Zeit als das lokale Traversieren der restlichen A Knoten, da A = n/3 - L/3. Die gesamten Kosten um den Baum zu traversieren betragen

$$\boldsymbol{\tau_3(n) = r + M + \tau_2(B) = 2L/3 + 2M + n/3}$$

<u>Allgemeiner Fall: P>2</u>

$\tau_{P-1}(B)$ = (P-2) L' / (P-1) + (P-2) M + B / (P-1), wobei L' die Länge des spines des Unterbaumes mit B Knoten ist. Der Offline-Algorithmus wählt r = L / P, sodaß B=(P-1)n / P und L'= (P-1)L / P.

Die gesamten Kosten sind demnach:

$\boldsymbol{\tau_P(n)} = r + M + \tau_{P-1}(B)$
$\quad = L/P + M + (P-2)(P-1)L/(P(P-1))+(P-2)M+(P-1)n/(P(P-1))$
$\quad = \mathbf{(P-1)L/P+(P-1)M+n/P}$.

Die Kosten des obigen Offline-Algorithmus ist also durch n/P + L + MP begrenzt.

□

<u>Theorem 5.1</u>

Das konkurrierende Verhältnis eines deterministischen Online-Algorithmus zu dem optimalen parallelen Algorithmus hat eine Untergrenze von $\frac{1}{3}\min\{P,n/L\}$; vorrausgesetzt wird, daß die Baumgröße $n > MP^2$ erfüllt.

Beweis

Die Kosten eines Online-Algorithmus liegen bei mindestens n Zeiteinheiten für einen T-Baum (nach Lemma 5.2). Die Kosten für den optimalen Offline-Algorithmus sind höchstens n/p+L+MP (nach Lemma 5.3). Es ist nicht bekannt, welche der drei Terme am größten ist, aber wir können die eigentlichen Kosten nach oben abschätzen mit 3 max{ n/p, L, MP}. Daher ist das Verhätnis der Laufzeit des Online- zum Offline-Algorithmus nach unten beschränkt durch $\frac{1}{3}\min\{P, n/L, n/MP\}$. Die Größe des dritten Term wird durch das n bestimmt und kann so beliebig groß gewählt werden. Der zweite Term muß nicht unbedingt mit n zunehmen, da auch L eine Funktion von n ist. Wenn nun die Eingabe-Bäume so eingeschränkt werden, sodaß $P \leq n/MP \Rightarrow n \geq MP^2$, dann ist die Grenze bestimmt durch die ersten zwei Terme: $\frac{1}{3}\min\{P, n/L\}$.

□

Das Theorem 5.1 impliziert, daß keine konkurrierender deterministischer Online-Algorithmus existiert, der das Granularity-Problem (entscheiden wann ein Task gespawned weden soll) löst. Das bedeutet, daß es für jeden Online-Algorithmus mindestens ein Baum existiert, den der Algorithmus sequentiell abarbeiten würde, während der optimale Algorithmus ihn fast vollständig parallel verarbeiten würde.

6. Experimentelle Meßergebnisse

Es werden einige Meßergebnisse des CG-Algorithmus vorgestellt.

	Power(22)		fib(32)		comb'(500,8)	

Gegeben sei ein deterministischer Online-Algorithmus. Ein Gegner kann einen T-Baum konstruieren, sodaß der Algorithmus mindestens M Zeiteinheiten für jeden vollbesetzten Unterbaum benötigt. Wenn der Algorithmus einen Task spawned, während er den Baum traversiert, dann benötigt allein schon das Spawning M Zeiteinheiten. Wird kein Task versendet, dann weiß der Gegner, wegen der deterministischen Traversierungsmethode, welcher Knoten als letztes besucht wird. Der Gegner plaziert dann genau an diesen Knoten den nächsten Unterbaum als Nachfolger. Das zwingt den Online-Algorithmus alle M Knoten des Unterbaumes zu besuchen, bevor die Wurzel des nächsten Unterbaumes erreicht wird.
Daher benötigt der Online-Algorithmus mindestens TM = n Zeiteinheiten für die Traversierung des gesamten T-Baumes.
□

Lemma 5.3
Es existiert ein Offline-Algorithmus, der einen T-Baum in weniger als n/P + L + MP Zeiteinheiten traversiert.

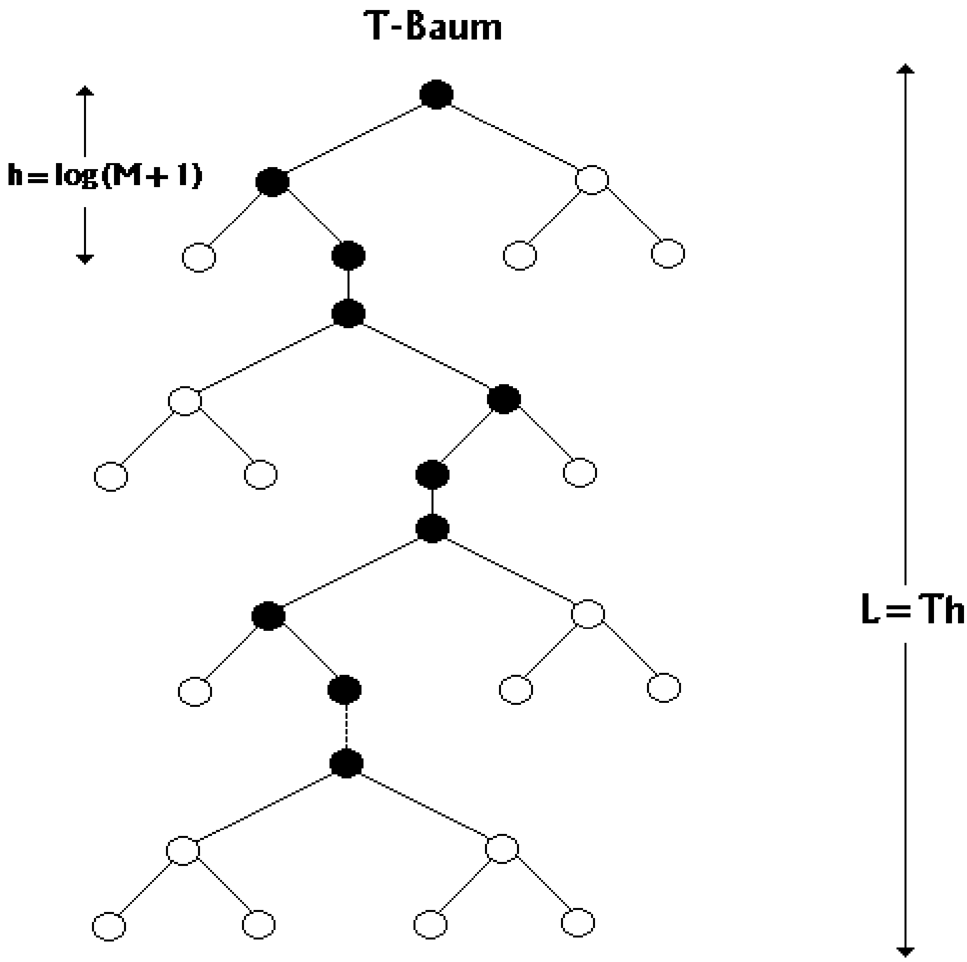
Beweis
Der Offline-Algorithmus hat den Vorteil, daß er die Struktur des Baumes a priori kennt. Der Offline-Algorithmus beginnt seine Traversierung (auf der ersten PE) an der Wurzel. Er traversiert entlang dem spine bis er an den r-ten Knoten gelangt („r wird gleich definiert“). Alle anderen entdeckten Tasks, die beim Spine-Traversieren entdeckt wurden, werden dem lokalen Pool hinzugefügt. Die erste PE sendet nun den gesamten Unterbaum (unter dem r+1ten Knoten) zu einer freien PE, um ihn dort parallel traversieren zu lassen. Die erste PE arbeitet zur selben Zeit an den restlichen Tasks aus dem Taskpool. Die PE, die den gesendeten Task empfängt benutzt für die Entscheidung eines weiteren Sendens das gleiche Prinzip.
Sei A die Anzahl der Knoten die für die lokale Traversierung vorgesehen sind, nachdem der Task (mit dem Unterbaum) einem anderen PE zugesendet wurde. Sei B die Anzahl der Knoten im gesendeten Task (der Unterbaum der den r+1ten spine-Knoten zur Wurzel hat). Die Anzahl der Knoten im T-Baum ist demnach n = A + B + r. Sei die Anzahl der PEs P > 1 und sei $\tau_P(m)$ die Laufzeit dieses Algorithmus, die er für das Verarbeiten eines T-Baumes mit m Knoten und unter Bereitstellung von P PEs benötigt.
Der Algorithmus traversiert entlang dem spine in Schritten der Größe h Knoten und versucht ein r zu wählen, sodaß $\tau_p(B) = A$, wobei p die Anzahl der freien PEs ist. Ein guter Wert für r ist die Höhe des Baumes multipliziert mit 1/p.
Um die Gleichungen des folgenden Beweises zu vereinfachen wird angenommen, daß alle notwendigen Werte sich ohne einen Rest teilen lassen.

Basis-Fall: Anzahl der PEs ist P = 2
Der Algorithmus setzt r = L / 2, sodaß B = n / 2. Aus n = A+B+r folgt, daß A < B, sodaß die Zeit den Task parallel zu verarbeiten größer ist als die Kosten für lokal verbleibenden Tasks.
Die gesamten Kosten sind demnach: $\boldsymbol{\tau_2(n) = r + M + B = L/2 + M + n/2}$.
(verursacht durch das sequentielle Traversieren von r spine-Knoten; dann das Spawnen eines Tasks, gefolgt vom sequentiellen Traversieren von B Knoten.)

Fall: P = 3
Der Offline-Algorithmus wählt r = L/3. Dadurch ist B = 2n/3. Der parallele Task benötigt unter Verwendung von 2 PEs (benutzen die Lösung vom Basis Fall)

$$\tau_2(B) = \frac{\frac{2L}{3}}{2} + M + \frac{\frac{2n}{3}}{2} = \frac{L}{3} + M + \frac{n}{3}$$

Die Höhe des T-Baumes ist demnach L = Th und die gesamte Knotenanzahl ist n = TM. Der Pfad, der von der Wurzel des T-Baumes bis zum untersten Level des T-Baumes führt wird ‘spine’ genannt.



<u>Lemma 5.2</u>
Für jeden deterministischen Online-Algorithmus existiert ein T-Baum, sodaß die Kosten für die Traversierung des Baumes mindestens n Zeiteinheiten benötigt. (Wobei n die Knotenanzahl des T-Baumes ist.)

Beweis

Angenommen es existiert ein konkurrierender deterministischer Online-Algorithmus der einen Baum, dessen Form a priori nicht bekannt ist, parallel traversiert. Die Laufzeit eines solchen Algorithmus sollte sich nur um einen kleinen konstanten Faktor von der des optimalen Algorithmus unterscheiden; und das für jeden Baum. Das folgende Beispiel zeigt, daß solch ein Algorithmus nicht existiert, zumindest wenn wir weiterhin noch fordern, daß die Konstante klein genug ist.

Einführung des T-Baumes

Ein T-Baum besteht aus T vollbesetzten binären Unterbäumen, die jeweils an der Spitze des nächsten Unterbaumes gesetzt werden (siehe Abb. 4). Der T-Baum ist so aufgebaut, daß jede Wurzel eines Unterbaumes (ausgenommen der oberste) ein Nachfolgeknoten eines Blattes des vorherigen Unterbaumes ist.

Jeder Unterbaum habe M Knoten, wobei M auch die Kosten für das Senden eines Tasks seien. Sei $h = \log(M+1)$ die Höhe des Unterbaumes.

5. Worst-Case Analyse

Kein konkurrierender Online-Algorithmus löst das Granularity-Problem

Es existiert kein konkurrierender Online-Algorithmus, der das Granularity-Problem löst. Das hat indirekt zur Folge, daß auch der CG-Algorithmus nicht als Lösung in Frage kommt. In der Tat existiert ein Fall, in der der CG-Algorithmus einen Baum sequentiell abarbeitet, obwohl der optimale Offline-Algorithmus ihn fast vollständig parallel verarbeiten würde. Der CG-Algorithmus nutzt also nicht jede sinnvolle Parallelität aus. Es gibt jedoch auch keinen anderen Online-Algorithmus, der es besser machen würde.

Obergrenze für Worst-Case-Laufzeit des CG-Algorithmus im Vergleich zum optimalen Offline-Algorithmus

Lemma 5.1

Das Competitive-Verhältnis zwischen dem CG-Algorithmus und dem optimalen Offline-Algorithmus hat eine Obergrenze (best-case) von 2 min{P, n / H}, mit P = Anzahl der PEs, H = Höhe des Baumes.

Beweis

Die Worst-Case Laufzeit des CG-Algorithmus bei einem Baum mit n Knoten ist (nach Lemma 4.1) 2n. Der optimale Offline-Algorithmus kann alle vorhandenen PEs verwenden; also hat er eine Untergrenze (worst-case) bei (n / P). Außerdem ist er auch durch die Höhe des Baumes nach unten begrenzt. Daraus folgt, daß der optimale Algorithmus eine Untergrenze von max{n/P,H} hat. Deswegen ist das Verhältnis der Kosten des CG-Algorithmus zu optimalem Offline-Algorithmus für jede Eingabe durch min{2P, 2n/H} = 2 min{P, n/H} begrenzt.

□

$$2M + \frac{(n+5)}{4}$$

Lemma 4.3
Im Fall eines vollbesetztem binären Baumes benötigt der CG-Algorithmus eine Laufzeit von:

$$(3M/2)\log P + \frac{(n-P+1)}{P}$$

Beweis
Die Kosten für das Verteilen der Tasks sind auch beim CG-Algorithmus MlogP, jedoch werden erst (M/2)logP lokale Operationen durchgeführt bevor ein Task versendet wird. (Der Faktor ist eher M/2 als M, da ein Task gespawned wird nachdem M Tasks aufgedeckt, aber nicht notwendigerweise besucht wurden.) Für die lokale Phase verbleiben (1/P) von dem restlichen Baum. Daher sind die Gesamtkosten für den CG-Algorithmus

$$(3M/2)\log P + \frac{(n-P+1)}{P}$$

□

Theorem 4.1
Im Fall eines vollbesetztem binären Baumes differiert die Laufzeit des CG-Algorithmus höchstens um Faktor 3/2 von der des optimalen Algorithmus.

Beweis
Das Verhältnis zwischen der Laufzeit des CG-Algorithmus (Lemma 4.3) und der des offline-Algorithmus (Lemma 4.2) ist immer weniger als 3/2 und geht gegen 1 für große n.
□

Beispiel:

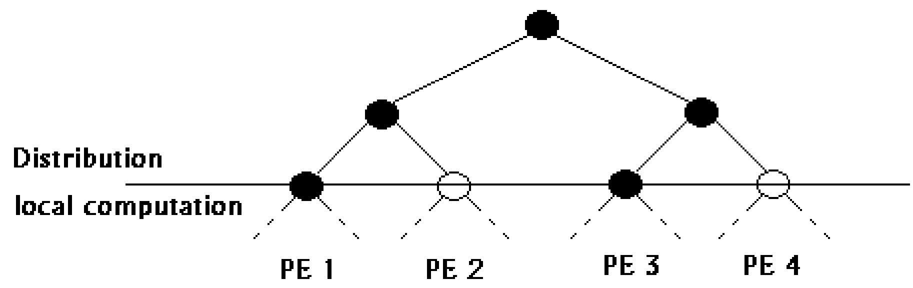


Abb.4 Der optimale Algorithmus angewendet auf einen vollbesetzten binären Baum unter Verwendung von 4 PEs

Zur Verdeutlichung des Lemma 4.2 eine optimale Ausführung des Algorithmus auf einen binären Baumes unter Verwendung von 4 PEs. Die gepunktete Linie bezeichnet das Ende der Verteilungsphase und somit den Beginn der lokalen Bearbeitungsphase. Beim Besuchen der Wurzel des Baumes werden zwei Unterbäume entdeckt. Der rechte Sohn wird an eine frei PE gesendet. Der linke Sohn wird besucht. Es werden wieder zwei Söhne entdeckt, von dem der Rechte wiederum an eine freie PE gesendet wird. Zur selben Zeit sendet auch der erste versendete Task, da auch er zwei Söhne entdeckt hat, seinen rechten Sohn an eine freie PE. Deswegen sind nun alle PEs am Arbeiten, jeder an einem viertel des restliches Baumes.

Die Kosten für die Verteilungsphase ist in diesem Beispiel 2 spawns und 2 traversierte Knoten. Alle anderen Spawnes und Traversierungen werden nicht gezählt, da sie parallel geschehen.
Die Kosten für die lokale Bearbeitungsphase wird berechnet, indem die traversierten Knoten aus der Verteilungsphase von n (Anzahl der Knoten im Baum) subtrahiert werden und dann durch 4 (Anzahl der PEs) teilen.
Dadurch benötigt der optimale Algorithmus in diesem Beispiel:

$$2M + \frac{(n+5)}{4}$$

Überprüfen des Ergebnisses:

$$(M+1)\log P + \frac{(n-P+1)}{P}$$

$$\Leftrightarrow$$

$$(M+1)\log 4 + \frac{(n-4+1)}{4}$$

$$\Leftrightarrow$$

$$2(M+1) + \frac{(n-3)}{4}$$

$$\Leftrightarrow$$

$$2M + 2 + \frac{(n-3)}{4}$$

$$\Leftrightarrow$$

Seien M die Kosten für das Spawnen eines Tasks. Der CG-Algorithmus spawned höchstens einen Task für M besuchte Knoten. Dadurch gilt die folgende Obergrenze für die Laufzeit des Algorithmus:

$$\left(\frac{n}{M}\right)M + n = 2n$$

Das entspricht den maximalen Kosten des Spawning, als Ergänzung zu den n Operationen die zum Traversiern der Knoten benötigt werden.

□

Korollar 4.1

Der CG-Algorithmus ist 2-competitive zu jedem sequentiellen Algorithmus.

Beweis

Das sequentielle Traversieren von n Knoten kostet n Zeiteinheiten. Nach Lemma 4.1 sind die Kosten für den CG-Algorithmus höchstens 2n. Deswegen differieren die Kosten des CG- und des Offline-Algorithmus um höchstens Faktor 2.

□

4.2 Fall: vollbesetzter binärer Baum

In diesem Fall soll jede mögliche Parallelität ausgenutzt werden, d.h. alle Tasks sind groß und sollen an freie PEs gespawned werden. Sei P die Anzahl der PEs und n die Anzahl der Knoten im Baum. Es sei n > PM.

Lemma 4.2

Bei einem vollbesetztem binärem Baum sind die Kosten für den optimalen parallelen Algorithmus:

$$(M+1)\log P + \frac{(n-P+1)}{P}$$

Beweis

Ein optimaler offline-Algorithmus führt einen solchen Baum in 2 Phasen aus:

Verteilung von Tasks und lokale Bearbeitung.

In der Verteilungsphase wird jeder erstellte Task zu einem freien PE gespawned, bis alle PEs besetzt sind. In der lokalen Phase bearbeitet jeder PE ein gleichen Teil des Baumes parallel. Bis eine Verteilung der Tasks parallel bewerkstelligt wurde benötigt man log P Schritte (alle PEs sind besetzt). Wobei jeder Schritt das Traversieren eines Knotens und das Senden eines Tasks beinhaltet. Daher sind die Kosten für die Verteilungsphase (M+1) log P. Wenn alle PEs besetzt sind, bearbeitet jeder den gleichen Teil (1 / P) des restlichen Baumes parallel. Da die Verteilungsphase einen binären Baum mit P Blättern bildet, ist die Gesamtzahl der traversierten Knoten P-1 und die Größe des restlichen Baumes n-P+1. Daher sind die Kosten für die lokale Bearbeitung (n-P+1)/P. Also betragen die Gesamtkosten des optimalen Algorithmus:

$$(M+1)\log P + \frac{(n-P+1)}{P}$$

□

4. Laufzeitanalyse

Es wird im folgenden, der CG-Algorithmus mit dem optimalen off-line Algorithmus für einige typischen Fälle verglichen. Der CG-Algorithmus arbeitet in realitätsnahen Situationen sehr effektiv.

Zuvor wird jedoch der '**c-competitive**' - Begriff eingeführt.
Ein Online-Algorithmus x ist zu einem anderen Algorithmus y c-competitive, wenn das worst-case Leistungsverhältnis von x und y, durch einen kleinen konstanten Faktor c begrenzt ist.
Diese Konstante c wird competitve ratio genannt.

4.1 Fall: sequentielle Lösung

Es wird die Klasse von Bäumen betrachtet, deren optimale Ausführung sequentiell verläuft. D.h. das jede Parallelität sollte vermieden werden, da die Größe eines jeden parallelen Tasks kleiner als die Kosten M für das Task-spawning wären.
In diesem Fall differiert die Laufzeit des CG-Algorithmus um maximal Faktor 2, und das für jeden Baum der mit dem Offline-Algorithmus sequentiell abgearbeitet werden würde.

<u>Lemma 4.1</u>
Die Kosten für das Traversieren eines Baumes mit dem CG-Algorithmus ist höchstens 2n, wobei n die Anzahl der Knoten des Baumes sind.

Beweis

erlaubt. Die Variable t wird um M dekrementiert. Dadurch wird garantiert, daß M Knoten lokal traversiert werden können, um den spawn zu bezahlen. Einige dieser Knoten wurden bereits traversiert, andere müssen noch traversiert werden. Wenn t immer noch größer als M ist, dann kann ein weiterer spawn stattfinden.
Zu beachten ist, daß t auch dekrementiert wird, wenn keine freie PE vorhanden ist. Auf diese Weise kann der Algorithmus vor einer Endlos-Schleife bewahrt werden.

3.2 Sourcecode

```
Const M; { cost of spawning a task, M > 0 }
Begin
        n := root node; { points at the current node }
        t := 0; { counts nodes in task-pool }

        Repeat
                While (t ≤ M) do
                        Begin
                                { visit node n }
                                add the children of n to task pool;
                                t := t + number of children of n;
                                n := get a node from the task pool;
                        End;
                If there is an idle PE
                        then Begin
                                        spawn one task from the task-pool to an idle PE;
                                        t := t - 1;
                                End;
                t := t - M;
        until no more nodes to visit;
End;
```

Abb.2 CG-Algorithmus

- Der linke Sohn wird als nächster besucht; hier enthüllen wir drei Söhne
- Der nächste Knoten (Blatt ganz links) entdeckt keine weiteren Knoten
- DFS-Traversierung besucht nächsten Knoten (Bruder-Knoten); ein Sohn wird entdeckt

In diesem Beispiel befindet sich ein Knoten, während der Traversierung, 'unberührt' im Task-Pool, nachdem die Wurzel besucht wurde. Also sollte dieser Knoten zu einer weiteren PE gesendet und dort parallel traversiert werden.

3. CG-Algorithmus

3.1 CG-Algorithmus

Der CG-Algorithmus versucht das garnularity-problem zu überwinden, indem er den Effekt des versehentlichen Sendens der Tasks, die kleiner als M (Kosten des spawning) sind, minimiert. Dies wird durch Ausgleichen der 'Menge der lokalen Arbeit' und den 'Kosten des spawning' erreicht. Genauer gesagt, wird dieses Gleichgewicht erlangt, indem sichergestellt wird, daß für jeden spawn eines parallelen Tasks noch M Knoten lokal besucht werden können. Dabei führt der Algorithmus jeden spawn in zwei Phasen durch:
<u>Traversierungs-Phase:</u>
Die Nachfolger-Knoten von jedem besuchten Knoten (Tasks) werden dem Task-Pool hinzugefügt. Wenn mindestens M+1 Knoten dem Task-Pool hinzugefügt wurden, dann wird die Allocation-Phase durchgeführt.
<u>Allocation-Phase:</u>
Wenn ein freier PE existiert wird ein Task aus dem Task-Pool gespawned.
Diese zwei Phasen werden iterativ wiederholt, solange der Baum traversiert wird.

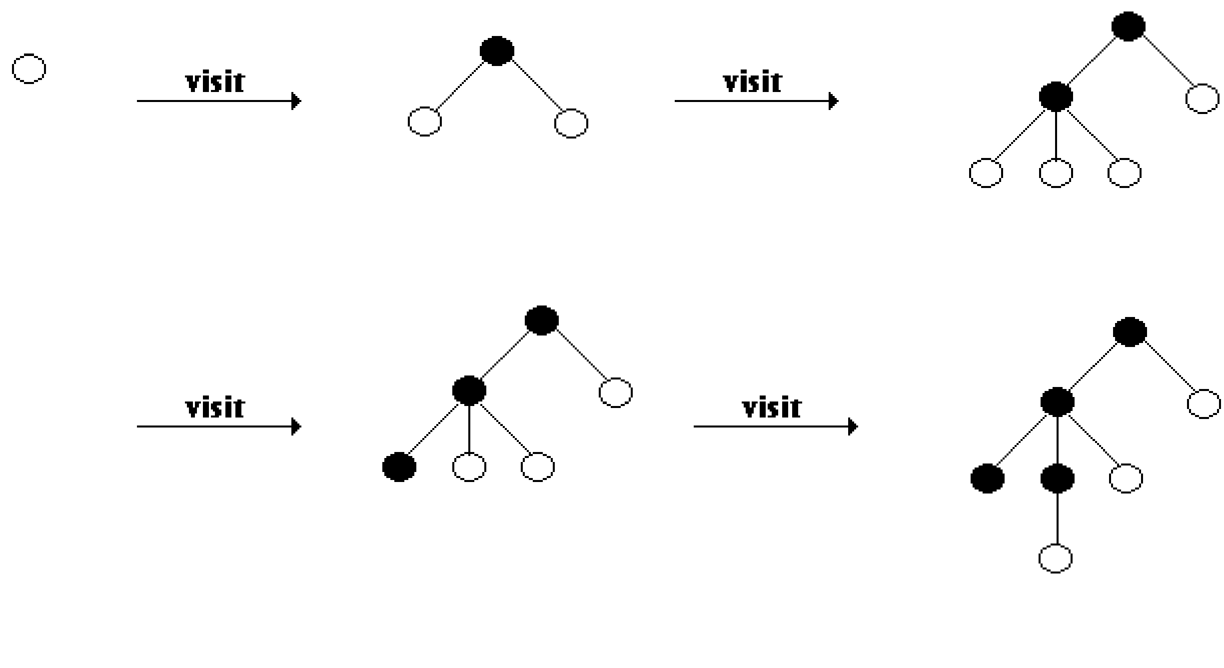
Zur Verdeutlichung des eigentlichen Algorithmus soll zuerst ein einfacheres Schema betrachtet werden, welches mindestens M Knoten lokal, für jeden gesendeten Task, besucht. Das kann aber zu einer unnötigen Verzögerung führen. Angenommen es seien k Knoten besucht worden, dann wären mehr als M-k Knoten im Task-Pool. In diesem Fall könnte ein Task sofort gespawned werden, vorausgesetzt daß M Knoten im Task-Pool lokal ausgeführt werden.
Das ist genau die Arbeitweise des CG-Algorithmus. Die Variable t ist ein Zaehler für die garantiert lokale Arbeit (dem Task-Pool hinzugefügte Knoten). Ist (t > M), dann ist ein spawn

Da die Anzahl der PEs in diesem Modell festgelegt ist, muß ein load-balancing Schema festgelegt werden, welches die Fälle behandelt in denen mehr Tasks als PEs zu Verfügungung stehen. Ein einfaches Schema wäre ein spawning eines Tasks nur dann zu zu lassen, wenn ein PE arbeitslos ist (also keinen Task mehr bearbeitet). Dazu muß nun also ein information-dissemination Schema entwickelt werden, das solche arbeitslosen PEs ausfindig macht. Das Model ist nicht auf ein bestimmtes Schema begrenzt, solange die Kosten in M enthalten sind.

Um die Analyse zu vereinfachen wird der Baum mit depth-first-search (DFS)-Traversierung durchlaufen und der älteste Task aus dem Task-Pool gewählt. Es wird später noch gezeigt, daß identische Ergebnisse erreicht werden für jede andere deterministische Traversierungs-Methode und Taskauswahl.

2.3 Beispiel

Um die Details des Modells zu veranschaulichen, ein kleines Beispiel:



Die gefärbten Knoten in Abb.1 repräsentierem die bereits Besuchten. Die hellen Knoten befinden sich im Task-Pool.

- Zuerst wird die Wurzel besucht; dabei werden zwei Nachfolger-Knoten entdeckt

In diesem Modell wird der *computation graph* durch einen Baum repräsentiert. Die Knoten entsprechen den Tasks und die Kanten zeigen die Abhängigkeiten der Tasks zueinander an. Das Aussehen und die Entwicklung des Baumes ist a priori nicht bekannt, da das Muster rekursiver Funktionen oder Schleifen im Allgemeinen schlecht vorrauszusagen ist. Das Bearbeiten eines Tasks wird durch den Besuch eines Knotens repräsentiert und kostet eine Zeiteinheit (Tasks die mehr als eine Zeiteinheit benötigen, können durch Kinder-Ketten dargestellt werden). Dieser Besuch des Knotens deckt auch die Nachfolger-Knoten (Kinder) auf, welche sich an den ausgehenden Kanten des besuchten Knotens befinden. Ziel ist es alle Knoten des Baumes parallel zu besuchen und das in minimaler Zeit.

Es wird ein Multicomputer-System mit verteiltem Speicher zugrunde gelegt, mit einer festen Anzahl an Prozessoren, welche über Nachrichten (Messages) kommunizieren. Auf solchen Systemen sind die Overhead-Kosten für das Versenden von Tasks relativ hoch, was die Grundidee des granularity problems nur unterstreicht.

Das Ausführen eines Programms in einer solchen Umgebung beginnt mit einer Prozessor Einheit (PE) die den Wurzel-Knoten des Baumes besucht. Dieser Besuch enthüllt die Nachfolger-Knoten, welche in einen lokalen Pool eingeordnet werden. Das Traversieren des Baumes wird durchgeführt, indem ein Knoten vom Task-Pool gewählt wird, der Knoten besucht wird und seine Nachfolgerknoten dem Pool hinzugefügt werden. Die Knoten (Tasks) im Pool sollen auch zu anderen PEs zum Traversieren (bzw. Ausführen) gesendet werden. Der Prozeß wird fortgeführt bis alle Knoten im Baum besucht wurden, also der lokale Task-Pool leer ist.

Sei M (Zeiteinheiten) die Kosten für das Senden. M ist ein system-abhängiger Parameter, welcher die Kosten für das Verpacken des Tasks in eine Nachricht, das Auffinden einer geeigneten PE, das Senden und das Empfangen des Tasks, Auspacken des Tasks am empfangenden Ende und das Senden und des Empfangen des Ergebnisses beinhaltet. Dieser Vorgang wird als „spawning“ bezeichnet. Es wird angenommen, daß die Kosten für das spawning M durch den Sender und dem Empfänger verursacht werden. D.h. wenn die PE A einen Task an die PE B sendet, A seine Arbeit nach einer Verzögerung von M fortsetzen kann und B seine Arbeit beginnen kann nach einer Verzögerung von M. Trotz dieser Vereinfachung, daß beide PEs die gleichen Kosten verursachen, ist dieses Modell realistischer als die meisten anderen, in denen nur der Empfänger die Kosten für die Kommunikation trägt. Denn erstens erfordert das Senden einige Arbeit (packen des Tasks in eine Nachricht, empfangen des Ergebnisses, usw.). Zweitens müßte bei synchronen Nachrichten der Sender auf ein acknowledgement warten. Und schließlich würde bei der Annahme eines kostenfreien Sendens beinahe alle parallelen Tasks zum „fernbearbeiten“ versendet werden, was aber aus praktischen Erfahrungen zu erheblichen Leistungsverringerungen führt.

Ein wichtiger Aspekt ist die minimal wirksame granularity (Menge der Arbeit) der gespawneden Tasks. Das Wesentliche des ganularity-problems ist die Beziehung zwischen der Anzahl der zu verarbeitenden Tasks (≈ Größe des Unterbaumes mit dem Task als Wurzel) und den Kosten für das spawning. Ein Leistungszuwachs kann garantiert werden, wenn die Menge der Arbeit des Tasks, welcher gespawned wird, größer als M ist. Und auch die verbleibende lokale computation muß größer als M sein. Da jedoch keine a priori Information über die Ausführungszeiten der Tasks bekannt sind, ist es unmöglich ein Schema zu entwickeln, welches nur ausreichend große Tasks spawned. Eine Alternative ist es die Tasks, die zu klein sind zu einem single Task zusammenzufassen. Das ist die zugrundeliegende Idee des später vorzustellenden CG-Algorithmus.

Teile zerlegt werden kann. Die Parallelisierung funktionale Programmiersprachen birgt noch weitere Schwierigkeiten, die es zu lösen gilt.

Funktionale Programme lassen sich leicht in eine große Menge kleiner Tasks aufteilen (→ da es ein fine-grained computation model ist). Jedoch hat eine zu feine Aufsplittung des Programms nicht unbedingt einen hohen Speedup zur Folge. Denn das System besitzt nur ein feste Menge an Prozessoren und es treten außerdem noch Kosten für das Erstellen und Senden der Tasks auf, welche leicht die Zeitersparnis der Parallelisierung dominieren kann (→ Kommunikations-Overhead). Daher kann es passieren, daß im worst-case ein paralleles Programm doppelt soviel Zeit als ein sequentielles benötigt. Es ist also besser eine gewisse Menge potentieller Tasks zu einem single Task zusammen zu fassen, um den Overhead möglichst klein zu halten.
Daher stellt sich die Frage, wann soll ein neuer Task versendet werden, damit die Parallelität hoch und der Overhead klein ist.

2. „Granularity Problem“

2.1 Granularity Problem

Während der Ausführung von parallelen funktionalen Programmen werden viele unabhängige Tasks erstellt. Einige von ihnen sollen zur parallelen Verarbeitung zu anderen Prozessor Elementen (PEs) versendet werden. Im Allgemeinen ist die Größe der einzelnen Tasks vor ihrer Ausführung nicht bekannt, sodaß eine Entscheidung darüber, ob ein Task nun lokal verarbeitet wird, oder zu relativ hohen Kosten fremdverarbeitet wird, schwer fällt.
Auf der einen Seite würde das Versenden eines kleinen Tasks den Kommunikations-Overhead unnötig erhöhen und so den Speedup verringern; auf der anderen Seite würde das verzögerte Senden eines großen Tasks eine Verringerung der Parallelität zur Folge haben.
Dieses Problem ist als das **„Granularity Problem“** bekannt.

2.2 Controlled Granularity (CG) Modell

Dieses Modell beinhaltet als Hauptsache die Entscheidung, wann ein Task gesendet werden soll. Aber es werden auch ein paar vereinfachende Annahmen getroffen, welche in der Realität nicht so vorgefunden werden. Z.B. werden die Kosten für das Versenden von Tasks als bekannt vorrausgesetzt; jedoch ist eine Schätzung der Kosten für das Rücksenden des Ergebnisses des Tasks, aufgrund der nicht bekannten Struktur, in realen Programmen schwer im Vorraus zu bestimmen. Desweiteren haben reale Programme oft komplexe Abhängigkeiten zwischen ihren Tasks.

1. Einleitung

1.1 Übersicht

- Es wird ein run-time Algorithmus vorgestellt, der ein Programm effektiv parallelisiert, falls dies möglich ist. Ineffektive Parallelität (durch zu viele kleine Tasks) wird ignoriert.
- Es wird das Verhalten des Algorithmus in unterschiedlichen Fällen analysiert und seine Laufzeit mit einem optimalem offline-Algorithmus verglichen.
  In den meisten Fällen erreicht der run-time Algorithmus vergleichbar gute Ergebnisse; obwohl der offline-Algorithmus Informationen über das zukünftige Verhalten der Kalkulation hat, jedoch der run-time Algorithmus nicht.
- Es wird bewiesen, daß es keinen konsequent besseren online-Algorithmus gibt.
- Es werden experimentelle Messergebnisse gezeigt, die die Effektivität des obigen online-Algorithmus demonstriert.

1.2 Problembeschreibung

Indem mehrere Prozessoren gleichzeitig an einem Problem arbeiten, werden höhere Verarbeitungsgeschwindigkeiten erzielt. Nimmt man an, daß p parallel arbeitende Prozessoren ein Problem bis zu p-fach schneller lösen als ein einzelner Prozessor, so ergibt sich bei einer Verknüpfung von 10^6 Prozessoren eine entsprechende Leistungssteigerung. Dieses Parallelverarbeitung setzt jedoch vorraus, daß das zu lösende Problem in viele gleichzeitig ablaufende

Inhaltsverzeichnis

J.W. Goethe Universität
Frankfurt am Main

Fachbereich Informatik (20) **Datum: 30.01.97**

Seminar WS 96/97

Parallelisierung und verteilte Ausführung funktionaler Programme

(Prof. Dr. Manfred Schmidt-Schauß)

A run-time algorithm for managing the granularity of parallel functional programs

von
Helmut Eckardt
eckardt@rbi.informatik.uni-frankfurt.de

ERROR: undefined
OFFENDING COMMAND: --------------620E511F6D9E--

STACK: