Portable parallele Baumsuchverfahren: Entwurf einer effizienten Bibliothek

Peter Sanders
LS Informatik für Ingenieure und Naturwissenschaftler
Univertsität Karlsruhe
D-76128 Karlsruhe
Email: sanders@ira.uka.de

Viele Probleme des Operations Research und der KI beruhen darauf, große Bäume und Graphen zu durchsuchen, die implizit (durch eine Knotenexpansionsfunktion) definiert sind. Diese Probleme sind oft sehr zeitaufwendig und deshalb ist es attraktiv, die Suche zu parallelisieren. Hier wird die Bibliothek PIGSeL beschrieben, die die u.U. recht aufwendige Parallelisierung von der Anwendung abkapselt. Gleichzeitig ermöglicht es dieser Ansatz, verschiedene Suchstrategien und parallele Maschinen mit den Anwendungen zu kombinieren. Es werden Entwurfsprinzipien vorgestellt, die helfen, die angestrebte Flexibilität in effizienter und benutzerfreundlicher Weise zu realisieren. Weiterhin wird ein Prototyp beschrieben, der parallele Tiefensuche mit einer effizienten dynamischen Lastverteilung auf einem Arbeitsplatzrechnernetz unter PVM und Transputersystemen unter dem Betriebssystem COSY realisiert.

1 Einführung

Ein entscheidendes Hindernis auf dem Weg zu einer breiten Verwendung von Parallelrechnern ist die Tatsache, daß die Entwicklung effizienter paralleler Programme erheblich aufwendiger sein kann als im sequentiellen Fall. Dies gilt insbesondere beim momentanen Stand der Entwicklungswerkzeuge und der Erfahrung der Programmierer. Erschwerend kommt hinzu, daß der erhebliche Mehraufwand an Zeit und Geld oft von neuem investiert werden muß, sobald die Anwendung auf die nächst- schnellere, bessere, neuere, größere Maschine portiert werden soll. Besonders problematisch können komplexe, unregelmäßig strukturierte Probleme sein, die sich nicht auf einfache daten-parallele Programme oder die bereits recht weit entwickelten Numerikbibliotheken zurückführen lassen. Ein typisches Beispiel für Algorithmen, die für irreguläre Probleme eingesetzt werden, sind Baumsuchverfahren wie Branch-and-bound, Spielbaumsuche oder Backtracking. Für diese gibt es eine Vielzahl von Anwendungen im Operations Resarch und der KI. Auch funktionale und logische Programmiersprachen lassen sich im weiteren Sinne als Baumtraversierungssysteme auffassen.

Hier soll nun der Entwurf der Bibliothek PIGSeL (Parallel Implicit Graph Search Library) beschrieben werden. PIGSeL versetzt den Benutzer in die Lage, eine parallele Anwendung zu erstellen, ohne sich um die Parallelisierung selbst kümmern zu müssen. Außerdem müssen nur die maschinenabhängigen Teile der Bibliothek an eine neue Maschine angepaßt nicht aber jede einzelne Anwendung. Es wird dargestellt, wie sich die Ziele Effizienz und Skalierbarkeit, breites Anwendungsspektrum, Anwenderfreundlichkeit und Portabilität in PIGSeL besser miteinander vereinbaren lassen, als dies in dem Autor bekannten existierenden Systemen möglich ist.

Abschnitt 2 ordnet den vorliegenden Ansatz zunächst in das Spektrum existierender Arbeiten zum Thema ein. Abschnitt 3 beschreibt dann den Entwurf der Bibliothek. Erfahrungen mit der Implementierung zweier Beispielanwendungen – des 0/1 Rucksackproblems und der Suche nach Golomblinealen – werden in Abschnitt 4 dargestellt. Abschnitt 5 faßt die Ergebnisse zusammen und gibt einen Ausblick auf in Zukunft noch zu untersuchende Fragestellungen. Im Anhang findet sich schließlich eine Zusammenstellung der wichtigsten Modulschnittstellen des Prototypen.

2 Grundidee und Einordung

Die Beschreibung von Suchverfahren durch generische Pseudocodefunktionen ist ein verbreitetes Mittel zur Einführung von Algorithmen in Lehrbüchern [1] oder zu ihrer Klassifikation und Exploration [2]. Abbildung 1 zeigt einen typischen generischen Branch-and-bound Algorithmus, der im folgenden wiederholt als Beispiel verwendet wird. Die Datenstruktur der abzuarbeitenden Knoten `agenda` wird mit der Wurzel des Suchbaums initialisiert. In der Hauptschleife wird jeweils ein Knoten n aus `agenda` ausgewählt und eine untere Schranke für eine aus n konstruierbare Lösung berechnet. Liegt die Schranke über der Bewertung der besten bisher bekannten Lösung, wird der Knoten weggeworfen; sonst stellt er entweder eine neue Lösung dar, oder er ist ein innerer Knoten, dessen Nachfolger in `agenda` eingefügt werden. Der Algorithmus findet alle Lösungen mit minimaler Bewertung.

Im Prinzip ist es nur ein kleiner Schritt vom Pseudocode zu einer realen Implementierung. Für Branch-and-bound sind z.B. abstrakte Datentypen „Agenda“ und „Suchbaumknoten“ einzuführen, die die konkrete Suchstrategie und die eigentliche Anwendung vom Suchalgorithmus abkapseln. Diese Modularisierung kann schon im sequentiellen Fall zur Wiederverwendbarkeit beitragen [3], denn der Anwendungsprogrammierer muß im wesentlichen nur Funktionen zum Erzeugen von Nachfolgeknoten zur Verfügung stellen. Aus den in der Einführung genannten Gründen ist dies aber für den parallelen Fall besonders verlockend [4, 5, 6]. In [4] wird eine generische parallele Tiefensuche beschrieben, bei der großer Wert auf eine mächtige Schnittstelle zur Anwendung und eine einfache Schnittstelle zur Maschine gelegt wird. Der Algorithmus ist in der Lage, mit dem Ausfall von Prozessoren zurechtzukommen. In [5, 6] wird generische Bestensuche (Best first search) auf Transputernetzwerken beschrieben. In [6] wird auch hybride Tiefen/Bestensuche berücksichtigt.

Die vorliegende Arbeit versteht sich dagegen weniger als eine Beschreibung eines bestehenden Systems, denn als eine Diskussion der Tradeoffs beim Entwurf. Es ist auch gar nicht das Ziel, irgendwann einmal zu einem abgeschlossenen monolithischen System zu gelangen. Vielmehr wird eine erweiterbare Sammlung sinnvoll miteinander kombinierbarer Module angestrebt.

```
agenda := {root node}
incumbent := ∞
WHILE agenda ≠ ∅ DO
    select some n ∈ agenda
    IF value(n) < incumbent THEN
        IF n is a leaf node THEN
            process new solution
            incumbent := value(n)
        ELSE
            insert successors of n into agenda
```

Abbildung 1: Typischer generischer Branch-and-bound Algorithmus

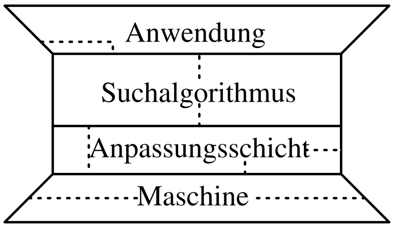

Abbildung 2: Grundstruktur der Bibliothek

3 Entwurf und Realisierung

Der hier beginnende Hauptteil dieser Arbeit diskutiert zunächst die prinzipielle Vorgehensweise (Abschnitt 3.1). Dann wird die in Abbildung 2 dargestellte Schichtenstruktur der Bibliothek von unten nach oben vorgestellt. Ganz unten gibt es verschiedene Typen abstrakter Maschinen (Hardware plus Betriebssoftware) deren Programmierschnittstelle durch eine Anpassungsschicht soweit vereinheitlich wird, daß ein Suchalgorithmus mit vielen Maschinen zusammenarbeiten kann (Abschnitt 3.2). Abschnitt 3.3 diskutiert Suchalgorithmen im allgemeinen und Abschnitt 3.4 behandelt spezifische Probleme der Parallelisierung wie Lastverteilung.

3.1 Prinzipielle Vorgehensweise

Die in Abbildung 2 angegebenen Schichtnamen müssen für den konkreten Entwurf noch präzisiert werden. Jede dieser Schichten kann aus mehreren Teilmodulen bestehen. (In der Abbildung durch gestrichelte Linien angedeutet.) Zum Beispiel kann bei dem Branch-and-bound Algorithmus aus Abschnitt 1 der Suchalgorithmus in Module für Ablaufkontrolle und Agenda unterteilt werden. Außerdem kann es für jedes (u.U. verfeinerte) Modul mehrere Implementierungen geben. Zum Beispiel kann die Agenda als Priority queue bzw. als Stapel implementiert werden und es ergibt sich Besten- bzw. Tiefensuche als resultierende Suchstrategie.

Genauer läßt sich dies unter Benutzung objektorientierter Terminologie formulieren: Den oben verwendeten Module entsprechen abstrakte Oberklassen einer Hierarchie von Unterklassen deren Blätter konkrete Implementierungen sind. Auch eine abstrakte Klasse kann bereits Code enthalten, allerdings baut dieser auf weiter unten in der Hierarchie implementierter Funktionalität auf.

Trotz der Attraktivität des objektorientierten Paradigmas benutzt PIGSeL einfaches ANSI-C, das (neben Fortran) die einzige Sprache ist, für die es auf den meisten Parallelrechnern eine einigermaßen ausgereifte und effiziente Implementierung gibt. Außerdem benötigt PIGSeL Objektorientierung nicht in ihrer vollen Allgemeinheit. Zu jedem Objekt ist zur Übersetzungszeit bekannt, zu welcher Klasse es gehört und nur eine einzige Implementierung einer abstrakten Klasse wird in einem gegebenen Programm verwendet. Diese eingeschränkte Form der Objektorientierung läßt sich relativ einfach (und sehr effizient) in C nachbilden.

3.2 Die Anpassungsschicht

Die Anpassungsschicht steht vor dem Dilemma, daß einerseits eine für alle Maschinen einheitliche Schnittstelle zum Suchalgorithmus angestrebt wird und andererseits die Stärken der Maschine dem Suchalgorithmus zugute kommen sollten. Ganz läßt sich dieses Problem wohl nicht lösen, denn Aspekte wie Multithreading, globale Synchronisation oder Eigenschaften des Verbindungsnetzwerks unterscheiden sich stark von Maschine zu Maschine und können einen starken Einfluß auf die verwendeten Algorithmen haben.

Trotzdem sind Kompromisse leichter als für eine allgemeine Standardbibliothek wie MPI [7], die einem viel breiteren Spektrum von Anwendungen gerecht werden muß. Dazu zwei Beispiele:

- Aus Portabilitätsgründen bietet MPI `broadcast` und andere kollektive Operationen nur in einer Variante, in der sich alle Prozessoren synchronisieren müssen. PIGSeL dagegen kann asynchrone kollektive Operationen auf jeder Maschine einigermaßen effizient realisieren, weil über die Interaktion Suchalgorithmus-Anpassungsschicht relativ viel bekannt ist.

- MPI besteht auf reihenfolgetreuer Auslieferung von Nachrichten, dies ist auf Maschinen, die Nachrichten um Engpäße herumleiten können, nur recht aufwendig zu realisieren. PIGSeL kann den darüberliegenden Schichten diese (oft gegenstandslose) Einschränkung „zumuten“ und kann damit effizienter sein.

Der in PIGSeL gewählte Ansatz besteht darin, die Anpassungsschicht selbst wieder zu modularisieren. Herzstück ist eine einfache Basisschnittstelle, die auf möglichst vielen MIMD-Rechnern realisierbar sein soll. Eine Zusammenfassung der Schnittstelle findet sich in Anhang B. Hier sollen nur die Grundideen dargestellt werden:

Jeder Prozessor führt das gleiche sequentielle C-Programm aus (SPMD Programmiermodell) und Prozessoren können asynchron Nachrichten an beliebige Kommunikationspartner senden. Durch entsprechende Routing software wie die von COSY [8] läßt sich dies selbst auf Transputern recht effizient realisieren. und selbst speichergekoppelte Maschinen sind für ein nachrichtengekoppeltes Programmierparadigma gut geeignet.

An einigen Stellen kommt es jedoch aufs Detail an. Zum Beispiel werden Handles (Nachrichtentypbezeichner) nicht als Konstanten definiert, sondern von einer Registrierungsfunktion vergeben. Zusätzlich kann für jeden Handle eine Funktion (Callback) festgelegt werden, die bei Empfang einer Nachricht des entsprechenden Typs automatisch aufgerufen wird. Dadurch können Module unabhängig voneinander agieren, ohne daß es zu Konflikten kommt. Sogar eine einfache Form von Multithreading läßt sich dadurch realisieren.

Ein gewisses Problem sind die von Maschine zu Maschine unterschiedlichen Konventionen zur Verwaltung von Nachrichtenpuffern. Dadurch gibt es für jede einfache und portable Schnittstelle Fälle, in denen eine Nachricht einmal öfter kopiert werden muß als nötig. In PIGSeL wurde der Ansatz von PVM [9] übernommen, Nachrichten durch Packmakros zusammenzusetzen, da dies eine sehr einfache und flexible Methode ist. Bisher wurde eine Schnittstelle für PVM und für Transputer unter dem Betriebssytem COSY [8] implementiert.

Zu dieser einfachen Schnittstelle hinzu kommen noch weitere Module, die zum Preis einer größeren Maschinennähe zusätzliche Funktionalität zur Verfügung stellen, die zur Vereinfachung und Beschleunigung der Suchalgorithmen beitragen kann:

- Module, für die portable Implementierungen existieren, die aber durch maschinenabhängige Reimplementierung beschleunigt werden können. Beispiel: Kollektive Operationen wie `broadcast` oder `reduceAdd`.

- Module mit einer maschinenunabhängigen Schnittstelle, die aber an jede Maschine angepaßt werden müssen. Zum Beispiel die Angabe eines Entfernungsmaßes zwischen zwei Prozessoren.

- Echt maschinenabhängige Operationen; z.B. Unterbrechungsbehandlung.

Je nachdem auf welche Art von Zusatzfähigkeiten ein Suchalgorithmus nun zurückgreift, entsteht ein einfach portierbarer, ein schwerer portierbarer oder ein überhaupt nicht portierbarer Suchalgorithmus. Die Anwendung selber bleibt davon aber unberührt.

```
agenda := {root node}
WHILE agenda ≠ ∅ DO
    select some n ∈ agenda
    insert successors of n into agenda
```

Abbildung 3: Baumtraversierungsalgorithmus.

3.3 Suchalgorithmen

Eine interessante Erfahrung mit dem Prototypen war, daß die direkte Übersetzung eines generischen Algorithmus subtile Effizienz- und Flexibilitätsprobleme mit sich bringt. Ein Problem ist z.B., daß Algorithmus 1 einen ganz bestimmten Zeitpunkt festlegt, an dem Knoten mit Hilfe der Variable `incumbent` beschnitten werden können. Eine auf Effizienz getrimmte Implementierung des Golomblinealproblems (Abschnitt 4) berechnet dagegen eine Approximation von `value` schon vor der Expansion, einen genaueren Wert danach, und in einigen Fällen werden sämtliche verbleibenden Nachfolger ohne weiteren Test beschnitten. Diese und alle denkbaren weiteren Varianten im generischen Algorithmus zu erfassen, würde zu einer hoffnungslos komplexen Schnittstelle zur Anwendung führen. Interessanterweise besteht die Lösung dieses Problems in einer Vereinfachung des generischen Algorithmus – sowohl die Anwendung von Heuristiken als auch das Erkennen neuer Lösungen wird der Anwendung überlassen. Was vom generischen Algorithmus bleibt, ist der in Abbildung 3 dargestellte Baumtraversierungsalgorithmus.[1] Zusätzlich werden nun allerdings folgende Regelungen benötigt:

- Wenn die Agenda als Priority queue organisiert ist, muß die Anwendung die Knotenbewertungsfunktion `value` weiterhin zur Verfügung stellen.
- Findet die Anwendung eine neue Lösung, so muß dies dem Suchalgorithmus durch Aufruf der Funktion `putNewSolution` mitgeteilt werden, damit die Lösung im parallelen Fall den anderen Prozessoren mitgeteilt werden kann.
- Umgekehrt muß die Anwendung die Funktion `getNewSolution` zur Entgegennahme einer neuen Lösung von außen anbieten.

Die Bibliothek hält gerade noch genug Fäden in der Hand, um die Parallelisierung abkapseln zu können. Außerdem stellt sich heraus, daß generische Algorithmen wie depth-first Branch-and-bound, IDA* und Backtracking für die Bibliothek ein und dasselbe sind: Baumtraversierung mit stapelförmiger Agenda. Wird stattdessen eine Priority queue als Agenda verwendet, kann damit best-first Branch-and-bound und eine einfache Variante von A* realisiert werden. Anhang A faßt die Funktionen zusammen, die implementiert werden müssen, um eine Anwendung zu erstellen.

Die oben beschriebenen Entwurfsentscheidungen legen den Einwand nahe, daß hier Benutzerfreundlichkeit zugunsten von Effizienz und Flexibilität zu kurz kommt. Das ist aber nicht ganz richtig. Zum einen ist es jederzeit möglich, ein kleines Hilfmodul anzubieten, das Knotenbeschneidung und Verarbeitung neuer Lösungen so abkapselt, daß die gleiche (inflexible) Schnittstelle entsteht wie beim Standardalgorithmus. Zum anderen bedeuted die

[1] Man ist versucht, diesen Ansatz ganz im Geist der Zeit mit dem Begriff *lean search* zu schmücken. Oder sollte gar die ganze Arbeit *lean libraries* genannt werden?

Selbstverwaltung von Heuristiken oft auch eine Vereinfachung. Zum Beispiel ist die Übersetzung von Maximierungs- in Minimierungsprobleme eine zwar triviale aber unangenehme und fehlerträchtige Angelegenheit.

Es gibt eine ganze Reihe weiterer Fälle, in denen sich herausstellt, daß die Übernahme einer Standardlösung zu Effizienzproblemen führt:

- Oft kann ein Suchbaumknoten sehr kompakt repräsentiert werden, wenn der Anwendung erlaubt ist, auf die Vorgänger eines Knoten zuzugreifen. (Zum Beispiel beim 0/1 Rucksackproblem.) Bei Tiefensuche ist dies nur eine Frage einer entsprechenden Schnittstelle zwischen Agenda und Anwendung. Bei Bestensuche erfordert dies aber zusätzliche Datenstrukturen.

- Selbst etwas so einfaches wie Tiefensuche hat viele Varianten:
  - Werden Knoten oder nur Unterschiede auf dem Stapel abgelegt? (Im Prototyp sind beide Varianten realisiert.)
  - Werden alle Nachfolger oder immer nur der aktuelle auf dem Stapel abgelegt?
  - Sollten Nachfolger sortiert werden? Und wenn ja, wer ist dafür zuständig?

- Es gibt eine große Zahl von hybriden Tiefen/Besten-Suchalgorithmen. Welche Variante ist wann günstig?

- Noch vielfältigere Möglichkeiten ergeben sich, wenn kompliziertere Algorithmen wie Spielbaumsuche oder Graphsuche betrachtet werden.

Insgesamt sind Patentlösungen also nicht zu erwarten. Trotzdem bietet Modularisierung einen gewissen Ausweg. Gelingt es, verschiedene Aspekte der Suchstrategie sauber zu trennen, so kann aus wenigen Grundbausteinen eine Vielzahl sinnvoller Kombinationen gebildet werden.

3.4 Parallelisierung

Die Diskussion im vorangegangenen Abschnitt hat gezeigt, daß allein die Unterschiede zwischen Anwendungen den Entwurf einer wiederverwendbaren Bibliothek schon hinreichend komplizieren. Sollen die Algorithmen auch noch parallelisiert werden, so hängt die beste Realisierung zusätzlich von der verwendeten Maschine ab. Typische Fragestellungen sind:

- Führt globale Kommunikation zu einer so guten Lastverteilung, daß der Mehraufwand gegenüber Nachbarschaftskommunikation lohnt?

- Kann spekulative Knotenexpansion zur Verbesserung der Knotenauslastung die Leistung verbessern.

- Ist der Algorithmus skalierbar?

- Ist statische oder dynamische Lastverteilung vorzuziehen?

Auch hier kann PIGSeL keine Patentrezepte bieten. Aber schon der Prototyp kann mit einer Parallelisierung aufwarten, die zumindest für Tiefensuche auf einem weiten Spekrum von Maschinen zu guten Ergebnissen führt. Beim *Random polling* Algorithmus [10, 11] arbeitet jeder Prozessor an einem eigenen Teilproblem. Im Fall von Tiefensuche ist dies ein durch einen Stapel definierter Teilbaum. Geht einem Prozessor a die Arbeit aus, so würfelt er eine Prozessornummer b und schickt eine Arbeitsanforderung an b; ist b selbst arbeitlos, kommt eine Ablehnung zurück und es wird erneut gewürfelt. Im Erfolgsfall spaltet b sein Teilproblem in zwei Teile und übermittelt einen an a. Ein wichtiger Freiheitsgrad dieses Algorithmus ist

die Methode zur Teilung des Suchbaums. Ein einfaches und recht robustes Verfahren besteht darin, den der Wurzel am nächsten gelegenen Knoten zu lokalisieren, von dem noch nicht alle Nachfolger expandiert sind. Von den verbleibenden Nachfolgern bleiben der 2., 4., 6., … am Ort und der 1., 3., 5., … wird abgegeben. In [11] wird Random polling eingehend analysiert. Einfach ausgedrückt stellt sich heraus, daß die Investition in globale Kommunikation auszahlt, da mit hoher Wahrscheinlichkeit deutlich weniger Kommunikationsoperationen nötig werden als bei den meisten bekannten Verfahren mit Nachbarschaftskommunikation.
Zusätzlich enthält der Prototyp folgende Komponenten:

- Ein Initialisierungsverfahren, das mit einem einzigen `broadcast` jeden Prozessor mit einem eigenen Stück Arbeit versieht [12, 11]. Damit wird eine mit Zeitverlust behaftete „Einschwingphase“ des Random polling Algorithmus vermieden.

- Ein Terminierungserkennungsprotokoll [13], das auf ringförmiger Kommunikation beruht. In Zukunft wird stattdessen eine effizientere Variante verwendet, die sich auf ein asynchrones `ReduceAdd` zurückführen läßt.

- Wird eine neue Lösung gefunden, so wird sie durch einen `broadcast` allen Prozessoren mitgeteilt. Werden oft neue Lösungen gefunden, kann dies zu einer Überlastung des Kommunikationsnetzes führen. Dies wird in Zukunft dadurch entschärft, daß die neue Lösung zunächst entlang eines Reduktionsbaums an Prozessor 0 gesendet wird, und nur wenn die Lösung tatsächlich die global beste ist, wird ein `broadcast` ausgelöst.

4 Beispielanwendungen

4.1 0/1 Rucksackproblem [1]

Gegeben sind n Gegenstände mit Preisen p_i und Gewichten w_i sowie ein Rucksack mit Kapazität M. Gesucht sind $x_i \in \{0, 1\}$, für die $\sum p_i x_i$ maximal wird unter der Nebenbedingung $\sum w_i x_i \leq M$. Eine klassische Branch-and-bound Lösung besteht darin, die Gegenstände nach ihrer Preisdichte p_i/w_i zu sortieren und mit den Gegenständen mit hoher Preisdichte beginnend eine Fallunterscheidung zu machen je nachdem ob $x_i = 1$ oder $x_i = 0$. Eine heuristische Bewertung beruht darauf, Gegenstände als teilbar anzunehmen.

Der entstehende Suchbaum ist tief und schmal, hat maximalen Verzweigungsgrad 2, ist sehr anfällig gegenüber Anomalien [14] und scheint für bestimmte Klassen von Problemen im Mittel polynomielle Größe zu haben. Ein Suchbaumknoten läßt sich durch 5 Zahlen codieren, wenn für die Konstruktion der abschließenden Lösung auf die Vorgängerknoten zugegriffen werden kann. Es wurde eine Möglichkeit entwickelt, die Knotenbewertung in $O(\log n)$ Zeit zu berechnen. Durch diese Feinkörnigkeit des Problems wirken sich „Reibungsverluste“ an den Schnittstellen deutlich aus. Eine handoptimierte sequentielle Version ohne Bibliothek war etwa 1.6-mal schneller als die Implementierung mit der Bibliothek. Und mehr als drei mal so schnell wie eine teilweise objektorientierte Version (Gnu C/C++ auf SPARC).[2]

4.2 Golomblineale [15]

Ein Golomblineal der Länge m mit n Markierungen ist ein Lineal mit Markierungen an den ganzzahligen Stellen $m_1, \ldots, m_n$ mit $0 = m_1 < m_2 < \cdots < m_n = m$ mit der Eigenschaft

[2] Dieses Ergebnis steht in markantem Gegensatz zur weit verbreiteten Auffassung, daß C++ höchstens 10–20 % langsamer ist als C. Sieht man sich die Benchmarks an, auf denen diese Ansicht beruht, stellt man jedoch fest, daß die zeitkritischen Teile gar nicht objektorientiert sind oder zumindest keine abstrakten Klassen verwenden. Möglicherweise läßt sich aus den hier betrachteten Problemen ein aussagekräftigerer Benchmark entwickeln.

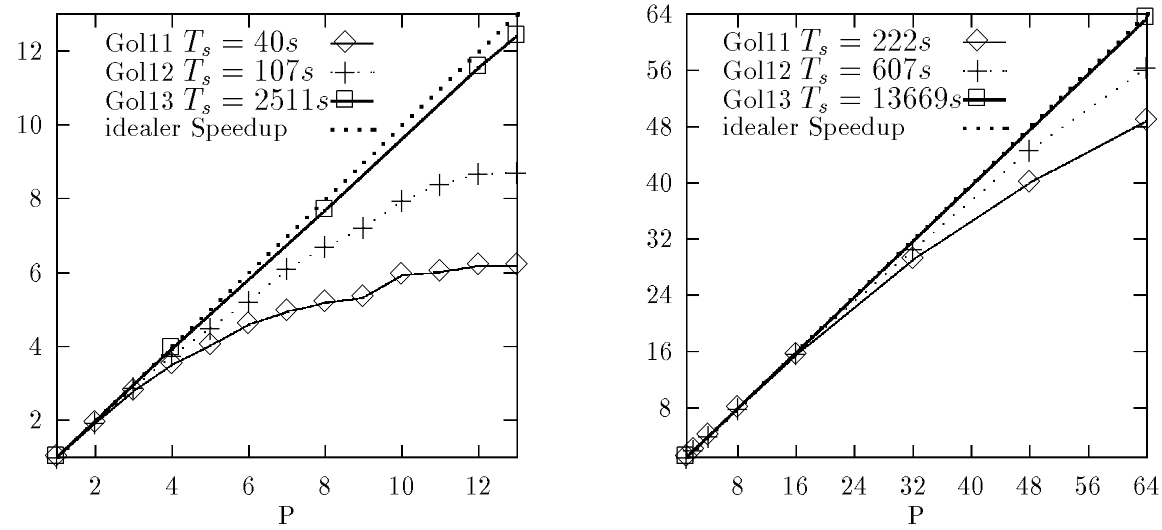

Abbildung 4: Speedup Kurven für PVM (links) und COSY (rechts) für Golomblineale mit 11, 12 und 13 Markierungen. T_s ist die sequentielle Ausführungszeit.

$|\{m_j - m_i : 1 \leq i < j \leq n\}| = n(n-1)/2$. Das heißt, ein Lineal, mit dem sich eine maximale Zahl Strecken ganzzahliger Länge messen läßt. Zu gegebenem n, $\overline{m}$ wird ein Golomblineal mit minimaler Länge $m \leq \overline{m}$ gesucht. Für dieses Problem gibt es Anwendungen in der Telematik und der Interferometrie. Hier wurde es jedoch hauptsächlich wegen seiner zum Rucksackproblem komplementären Eigenschaften ausgewählt. Der entstehende Suchbaum ist relativ breit und flach. Knotenauswertungen sind aufwendiger und deshalb sind Reibungsverluste an den Schnittstellen unerheblich. Dafür ist die Beschaffenheit der Schnittstelle um so wichtiger. Im Laufe der Entwicklung wurde durch verbesserte Heuristiken eine Beschleunigung um einen Faktor 129 erzielt. Die meisten Verbesserungen lassen sich aber nur realisieren, wenn die Anwendung die Beschneidung von Knoten selbst in der Hand hat.

Abbildung 4 zeigt Speedup Kurven für ein Ethernet mit bis zu 13 SPARC-SLC Arbeitsplatzrechner und PVM 3.2 sowie einen Parsytec Supercluster mit bis zu 64 als Torus vernetzten T800 Transputern mit 25 MHz Takt unter dem Betriebssystem COSY. Es wurden Probleminstanzen verwendet bei denen $\overline{m}$ so klein gewählt wird, daß keine Lösungen gefunden werden; dadurch werden Speedup Anomalien ausgeschlossen.

Es zeigt sich, daß die Lastverteilungstrategie Random polling in der Lage ist, Tiefensuche mit geringem Overhead auf sehr unterschiedlichen Plattformen zu parallelisieren. Allerdings müssen für ein lose gekoppeltes System wie ein Ethernet deutlich größere Probleme betrachtet werden, um einen guten Speedup zu erhalten.

5 Zusammenfassung und Ausblick

In den vorangehenden Abschnitten wurde am Beispiel paralleler Baumsuchverfahren untersucht, welche Aspekte zu beachten sind, um zu einer effizienten, flexiblen und portablen Bibliothek für die Parallelisierung irregulärer Probleme zu gelangen. Es stellt sich heraus, daß eine Vielzahl von Aspekten auf allen Ebenen des Entwurfs zu beachten sind. Patentlösungen, die die Parallelisierung eines generischen Algorithmus wie Branch-and-bound ein für allemal lösen, sind deshalb nicht zu erwarten. Andererseits gibt es oft überraschend einfache Teillösungen, die für eine große Klasse von Fällen zu guten Ergebnissen führen. Beim Prototypen beobachtete Beispiele dafür sind die SPMD Schnittstelle, die einfach zu benutzen und auf vielen Rechnern effizient realisierbar ist; der extrem einfache generische Baumtraversierungsalgorithmus und das Random polling Lastverteilungsverfahren, das zumindest für

Tiefensuche auf ganz unterschiedlichen Maschinen zu guten Ergebnissen führt. Ferner ist es durch ein konsequentes modulares Design möglich, schrittweise komplexeren Aufgabenstellungen gerecht zu werden, ohne jedesmal von vorne zu beginnen.

Gegenwärtig ist eine neue Version der Bibliothek in Entwicklung, die auf allen Ebenen breiter angelegt ist. Das Spektrum der betrachteten Maschinen soll erweitert werden (z.B. KSR-1); die Anpassungsschicht wird komplexere Lastverteilungsstrategien zu unterstützen haben und es werden neue Algorithmenklassen wie Spielbaumsuche oder Graphsuche betrachtet werden.

Anmerkungen

[1] E. Horowitz und S. Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, 1978.

[2] D. S. Nau, V. Kumar und L. Kanal. General branch and bound, and its relation to A* and AO*. *Artificial Intelligence*, 23:29–58, 1984.

[3] R. V. Johnson. Efficient modular implementation of branch-and-bound algorithms. *Decision Sciences*, 19(1):17–38, 1988.

[4] R. Finkel und U. Manber. DIB— A distributed implementation of backtracking. *ACM Trans. Prog. Lang. and Syst.*, 9(2):235–256, Apr. 1987.

[5] G. P. McKeown, V. J. Rayward-Smith und S. A. Rush. Parallel branch-and-bound. In *Advances in Parallel Algorithms*, Seiten 349–362. Blackwell, 1992.

[6] N. Kuck, M. Middendorf und H. Schmeck. Generic branch-and-bound on a network of transputers. In R. Grebe u.a., Herausgeber, *Transputer Applications and Systems*, Seiten 521–535. IOS Press, 1993.

[7] MPI Forum. MPI: A message-passing interface standard. Technical report, University of Tennessee, May 1994.

[8] R. Butenuth und S. Gilles. COSY — ein Betriebssystem für hochparallele Computer. In *Transputer Anwender Treffen*, 1994.

[9] A. Geist u.a. PVM 3.0 users's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, 1993.

[10] V. Kumar, A. Grama, A. Gupta und G. Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms*. Benjamin/Cummings, 1994.

[11] P. Sanders. Analysis of random polling dynamic load balancing. Technical Report IB 12/94, Universität Karlsruhe, Fakultät für Informatik, April 1994.

[12] O. I. El-Dessouki und W. H. Huen. Distributed enumeration on between computers. *IEEE Transactions on Computers*, C-29(9):818–825, September 1980.

[13] E. W. Felten. Best-first branch-and-bound on a hypercube. In *Proceedings of the 3rd Conference on Hypercubes, Concurrent Computers and Applications*, Seiten 1500–1504, Pasadena, 1988.

[14] A. P. Sprague. Wild anomalies in parallel branch and bound. Technical Report CIS-TR-91-04, CIS UAB, Birmingham, AL 35294, 1991.

[15] G. S. Bloom und S. W. Golomb. Applications of numbered undirected graphs. *Proceedings of the IEEE*, 65(4):562–570, April 1977.

A Von der Anwendung zu unterstützende Funktionalität

```
typedef ... Node;
```
Suchbaumknoten
```
typedef ... Difference;
```
Unterschied zwischen zwei Suchbaumknoten.
```
typedef ... Solution;
```
Lösung
```
void applInit(Node *root);
```
Anwendung initialisieren. Liefert Zeiger auf Wurzelknoten.
```
void apply  (Node *this, Difference d);
void unapply(Node *this, Difference d);
```
Ersetze `this` durch den durch `d` definierten Nachfolger/Vorgänger von `this`.
```
int          moreSuccessors(Node *this);
Difference   firstSuccessor (Node *this);
Difference    nextSuccessor (Node *this);
Difference currentSuccessor (Node *this);
```
Iterator über die Nachfolger eines Knotens.
```
void   getNewSolution(Solution *sol);
```
Nimm eine neue Lösung zur Kenntnis.
```
void printNewSolution(Solution *sol);
```
Gib neue Lösung aus.

B Von der Anpassungsschicht zu unterstützende Funktionalität

```
int nProc;
```
Gesamtzahl Prozessoren.
```
int iProc;
```
Lokale Prozessornummer (0 bis `nProc` $- 1$).
```
typedef void (*Callback)(void);
```
Funktionen, die bei Nachrichtenempfang aufgerufen werden.
```
int newHandle(Callback f);
```
Erzeuge einen neuen Nachrichtentyp, registriere `f` als Handler und liefere Handle zurück.
```
void initSend(int handle);
```
Bereite das Senden einer Nachricht vom Typ `handle` vor.
```
void send(int to);
```
Schicke fertige Nachricht an Prozessor `to`.
```
void   receive(void);
int tryReceive(void);
```
Synchrones/versuchendes Empfangen.
`tryReceive` $= 1$ gdw. eine Nachricht vorhanden ist.
```
void spmdPack  (void *p, int len);
void spmdUnpack(void *p, int len);
```
Ein/auspacken von Daten in/aus Nachrichtenpuffer.
```
double elapsedTime(void);
```
Lokale reale Zeit seit Programmstart in Sekunden.
```
int msgHandle;
int msgLen;
int msgSender;
```
Daten der zuletzt empfangenen Nachricht: Typ, Länge und Absender.