BMBF

Verbmobil
Verbundvorhaben

Parallelisierung eines inkrementellen aktiven Chart-Parsers

Jörg Spilker

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Vm REPORT 105
Dezember 1995

Dezember 1995

Jörg Spilker

Institut für Mathematischen Maschinen und Datenverarbeitung VIII
Lehrstuhl für Künstliche Intelligenz
Friedrich-Alexander-Universität Erlangen-Nürnberg
Am Weichselgarten 9
91058 Erlangen

Tel.: (09131) 8599 - 13
Fax: (09131) 8599 - 05
e-mail: spilker@immd8.informatik.uni-erlangen.de

Gehört zum Antragsabschnitt: 15.7 Architektur integrierter Parser für gesprochene Sprache

Die vorliegende Arbeit wurde im Rahmen des Verbundvorhabens Verbmobil vom Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie (BMBF) unter dem Förderkennzeichen 01 IV 101 G gefördert. Die Verantwortung für den Inhalt dieser Arbeit liegt bei dem Autor.

INHALTSVERZEICHNIS

KURZFASSUNG

Die vorliegende Arbeit beschreibt die Parallelisierung eines Chart-Parsers zur Syntaxanalyse gesprochener Sprache auf einem Rechner mit *Shared-Memory*. Ein besonderes Anliegen ist dabei die Integration in ein inkrementell arbeitendes Sprachverarbeitungssystem. Eine zentrale Koordinationsstelle, die Agenda des Chart-Parsers, verwaltet Aufträge, die parallel ausgeführt werden können. Die Leistungsfähigkeit des Parsers wurde unter realen Bedingungen getestet. Ein nenenswerter Geschwindigkeitsvorteil ist nicht zu verzeichnen. Die Gründe sind zum großen Teil auf eine nicht optimale Parallelisierung zurückzuführen, die sich aus der Erweiterung eines bestehenden sequentiellen Systems ergibt. Es werden aber auch prinizpielle Einschränkungen des parallelen Chart-Parsings in einer inkrementellen Architektur herausgearbeitet.

1 EINLEITUNG

Die Entwicklung paralleler Rechnerarchitekturen begann in den sechziger Jahren. Standen am Anfang numerische Anwendungen im Vordergrund, so erweiterte sich das Anwendunggebiet in den achtziger Jahren, vor allem aufgrund der Verbreitung von Multiprozessorrechner, auf andere Bereiche in denen hohe Rechenleistungen gefordert werden.

Eines dieser Anwendungsbereiche ist die Verarbeitung gesprochener Sprache. Das Ziel dabei ist ein technisches System, das fließend gesprochene Sprache in Echtzeit „versteht“ und entsprechend ihrem Inhalt reagiert. Der Kern eines solchen Systems wird klassisch in die drei linguistischen Abstraktionsebenen Ordnung, Inhalt und Gebrauch eingeteilt. Die Ordnung stellt Regeln zur Kombination von Symbolen dar. Diese Ordnung betrifft Unterebenen Phonolgie, Morphologie und Syntax. Beim Inhalt (Semantik) wird die Bedeutung der Symbole untersucht. Der Gebrauch umfaßt die Unterebenen der Pragmatik und des Diskurses. Er untersucht die Bedeutung der Symbole im aktuell gegebenen Kontext. Zu diesen linguistischen Ebenen kommen noch die Schallanalyse und die Schallerzeugung hinzu. Aus technischer Sicht können diese Ebenen und Unterebenen als eigenständige Softwaremodule gesehen werden, die über bestimmte Kommunikationskanäle verbunden sind. Die traditionelle Kopplung ist dabei eine sequentielle Hintereinanderschaltung aller Module. Jedes Modul reicht nur das endgültige Ergebnis seiner Analyse weiter. Eine vor allem psycholinguistsisch motivierte Parallelisierung ist eine inkrementelle Architektur. Alle Module arbeiten gleichzeitig und geben Zwischenergebnisse so früh wie möglich weiter.

Einige Module selber sind jedoch von einer Komplexität, die es sinnvoll erscheinen läßt auch innerhalb der Module parallele Abläufe zu definieren. Ansätze dazu sind vor allem in der Syntaxanalyse getestet worden, die eines der besterforschten Teilgebiete der Sprachverarbeitung ist. Die bisherigen Implementierungen sich dabei aber meist auf kleine Beispiele mit Texteingabe beschränkt und orientieren sich an der sequentiellen Kopplung der Module. Das Ziel dieser Arbeit ist die Evaluation eines inkrementellen, parallelen Parsers unter realen Bedingungen am Beispiel eines Zugauskunftssystems. Ausgangspunkt der parallelen Syntaxanalyse ist ein Chart-Parser. Eine sequentielle Version des Parsers, implementiert in LISP, stand schon zu Beginn der Arbeit zur Verfügung. Das verwendete LISP ist aber nicht zur Parallelverarbeitung geeignet. Für die Parallelisierung des Parsers mußte daher zuerst die Programmiersprache um entsprechende Konstrukte ergänzt werden.

Diese beiden unabhängigen Aufaben bestimmen den Aufbau dieser Arbeit. Im nächsten Kapitel werden für die weiteren Kapitel wichtige Grundlagen geschaffen. Kapitel drei beschäftigt sich mit einigen ausgewählten Problemen der LISP-Parallelisierung. Im Kapitel vier wird dann ein abstrakter Algorithmus für einen Chart-Parser angegeben, der die Grundlage der im fünften Kapitel beschriebenen Parallelisierung ist.

2

PARALLELE PROGRAMMIERUNG

Die Entwicklung von Computerprogrammen ist geprägt durch die vorherrschende sequentielle Rechnerarchitektur. Parallelrechner sind noch die Ausnahme. Die Entwicklungen im Bereich der Betriebssysteme[1] zeigen aber, daß sich parallele Ansätze langsam durchsetzen. Die Benutzung fortschrittlicher, objektorientierter Programmierstile erleichtert dabei den Übergang von einer sequentiell- zu einer parallel-orientierten Denkweise.

2.1 Prinzipielle Überlegungen zur parallelen Programmierung

Momentan werden noch viele verschiedene Ansätze zur Entwicklung und Programmierung paralleler Rechnerarchitekturen verfolgt. In welche Richtung diese Entwicklung geht, ist zur Zeit noch nicht absehbar. Unabhängig von den verschiedenen Ansätzen lassen sich jedoch einige prinzipielle Überlegungen zur Entwicklung eines parallelen Systems anstellen. Im folgenden wird dabei nur auf die Programmierung eingegangen. Die angesprochenen Punkte lassen sich aber zum großen Teil auch auf die Rechnerentwicklung übertragen.

2.1.1 Begriffsklärung

Die in der Literatur zu paralleler Programmierung auftretenden Begriffe wie Programm, Prozeß, Thread etc. werden leider nicht einheitlich definiert. Daher ist eine informelle Einführung der in dieser Arbeit verwendeten Begriff notwendig. Die Definition der Begriffe hält sich dabei eng an ihre Verwendung innerhalb des SOLARIS Betriebssystems.

Teilt man die Beschreibung einer Problemlösung in die drei von Marr in [Mar82] vorgeschlagenen Ebenen algorithmische Theorie, Repräsentation, Implementation ein, so lassen sich die verwendeten Begriffe durch die Ebene, auf der sie eingeführt werden, definieren. Zusätzlich wird eine vierte Ebene, die Hardware eingeführt.

[1] Z.B. die Spezifikation einer POSIX-Threadschnittstelle oder das Threadkonzept von Windows NT.

- **Algorithmische Theorie**

  Sie beschreibt die grundlegenden Eigenschaft, die jeder Ansatz zur Lösung des betrachteten Problems erfüllen muß. Es wird festgestellt, was berechnet werden muß und welche Einschränkungen von dem Problem vorgegeben sind. Auf dieser Ebene lassen sich z.B. Aussagen über die Komplexität des Problems machen. Aussagen über Parallelisierung sind hier noch nicht möglich.

- **Repräsentation**

  In dieser Ebene werden die notwendigen Datenstrukturen und der Algorithmus zu ihrer Verarbeitung definiert. Die hierbei mögliche Parallelität wird als **logische** oder **algorithmische Parallelität** bezeichnet.

- **Implementation**

  Die Implementation realisiert den Algorithmus in einer bestimmten Programmiersprache. Auf dieser Ebene müssen alle verwendeten Datenstrukturen, also sowohl die für den Algorithmus notwendigen als auch weitere Hilfsstrukturen genau spezifiziert werden. Die Parallelität der Implementation wird durch **Threads** beschrieben. Abhängigkeiten verschiedener Datenstrukturen untereinander können dazu führen, daß die algorithmische Parallelität nicht vollständig auf Threads abgebildet werden kann. Das Ergebnis der Implementation ist ein **Programm**.

- **Hardware**

  Prozessoren stellen die real vorhandene Hardware dar. Wird ein Programm auf einem Prozessorsystem ausgeführt, so stellt dieses einen **Prozeß** dar. Die Anzahl gleichzeitig innerhalb des Prozesses ausgeführter Threads wird als **echte Parallelität** bezeichnet.

2.1.2 Kenngrößen

Der entscheidende Vorteil eines parallelen Algorithmus soll in der Regel die höhere Geschwindigkeit gegenüber der sequentiellen Version sein. Dieser **Speed-Up** S wird bestimmt durch das Verhältnis der Laufzeit[2] des sequentiellen Algorithmus T(1) zur Laufzeit des parallelen Algorithmus mit P Prozessoren T(P).

$$S(P) = \frac{T(1)}{T(P)}$$

T(1) ist hierbei die Zeit für den besten bekannten sequentiellen Algorithmus. Der maximale theoretische *Speed-Up*[3] bei der Verwendung von P Prozessoren liegt also bei P. Setzt man den erzielten *Speed-Up* ins Verhältnis zu der Zahl der eingesetzten Prozessoren, so erhält man die **Effizienz** E des Algorithmus.

$$E(P) = \frac{S(P)}{P}$$

Die Effizienz zeigt also an, welcher Anteil der Rechenzeit für die Problemlösung genutzt wird. Sie gibt damit den Wirkungsgrad des parallelen Systems an.

[2] T gemessen in Zeitschritten, d.h. in einer Zeiteinheit wird genau eine Operation ausgeführt.

[3] Auf superlineare *Speed-Ups* und ihre Gründe soll hier nicht näher eingegangen werden. Siehe dazu [Bre92].

2.1.3 Mögliche Effizienzverluste

In einer realen Anwendung wird der maximale *Speed-Up* kaum zu erzielen sein. Damit verringert sich auch die Effizienz. Dieser Effizienzverlust hat nach [Bre92] mehrere Gründe:

- unvollständige Parallelisierung

  Normalerweise enthält ein Algorithmus Teilaufgaben, z.B. die Initialisierungsphase, die nicht parallelisierbar ist. Amdahl[Amd67] zerlegt daher die Gesamtlaufzeit T(1) in eine Laufzeit für den inhärent sequentiellen Teil T_s und eine Laufzeit für den parallelisierbaren Teil T_p.

  $$T(1) = T_s + T_p$$

  Normiert man T(1) auf 1, gilt:[4]

  $$T_p = 1 - T_s.$$

  Unter der Annahme, daß T_p beliebig oft aufgeteilt werden kann, ist

  $$T(P) = T_s + \frac{1 - T_s}{P}.$$

  Für S(P) ergibt sich dann

  $$S(P) = \frac{1}{T_s + \frac{1-T_s}{P}} \leq \frac{1}{T_s} \qquad \textbf{(Amdahls Gesetz).}$$

  Der inhärent sequentielle Anteil eines Algorithmus bestimmt unabhängig von der Anzahl der verfügbaren Prozessoren eine obere Schranke für den *Speed-Up*. Selbst der Einsatz mehrerer hundert Prozessoren bringt also bei einem zehnprozentigen Anteil von T_s an der Gesamtlaufzeit nur maximal eine Geschwindigkeitssteigerung um den Faktor zehn.

- algorithmische Verluste

  Durch die Veränderung des Algorithmus können sich zusätzliche Berechnungen ergeben. Zum Beispiel ist es möglich, daß der Wert eines Ausdrucks von jedem Prozessor einzeln ermittelt wird, um Kommunikationszeit zu sparen. Als Kenngröße definiert Hoßfeld in [Hoß83] die **Operationsredundanz** $R(P)$:

  $$R(P) = \frac{Z(P)}{Z(1)}.$$

  Z(1) ist hierbei die Zahl der Operationen des sequentiellen Algorithmus und Z(P) die Gesamtzahl der von P Prozessoren ausgeführten Operationen des parallelen Algorithmus. Definiert man die **Auslastung** U des Gesamtsystems als

  $$U(P) = \frac{Z(P)}{P * T(P)},$$

  so läßt sich die Effizienz auch als

  $$E(P) = \frac{U(P)}{R(P)}$$

  beschreiben. Damit wird sofort der Einfluß der Operationsredundanz auf die Effizienz sichtbar. Je höher die Redundanz eines Algorithmus bei einem voll ausgelastetem System (U(P)=1) ist, desto geringer ist seine Effizienz.

[4] Dies ist ohne weiteres möglich, da für S(P) und E(P) nur das Verhältnis von T(1) zu T(P) interessant ist.

- Kommunikations- und Synchronisationsverluste

  Kommunikation ist der Austausch von Daten zwischen Threads, während die **Synchronisation** den korrekten Ablauf des parallelen Programms sicherstellt. Beide Begriffe werden oft als äquivalent angesehen, da sie sehr eng zusammenhängen. Kommunizieren z.B. zwei Threads in einem *Shared-Memory* System über gemeinsam genutzte Variablen, so stellt die Synchronisation einen geregelten Zugriff auf diese Variablen sicher.[5]

  Der Idealfall völlig unabhängiger Threads wird bei einer parallelen Problemlösung selten erreicht. In der Regel müssen an bestimmten Programmstellen Zwischenergebnisse ausgetauscht oder auf das Ende von Berechnungen gewartet werden. Die Zeit, die für diese Aufgaben benötigt wird, wird unter dem Begriff **Kommunikationszeit** zusammengefaßt. Die Gesamtlaufzeit T(P) setzt sich somit zusammen aus der sich durch die Parallelisierung ergebenen Rechenzeit T_R und der Kommunikationszeit T_K.

$$T(P) = T_R + T_K$$

  Für Fall der **barrier Synchronisation**, d.h. alle Prozessoren warten an einer Komunikationsstelle, lassen sich T_R und T_K einfach ermitteln. Das Beispiel[6] einer parallelen Auswertung des Ausdrucks ((3+4)*(100+120))*((20+30)*(10+5)) mit zwei Prozessoren, wie es in Abb. 2.1 dargestellt ist, soll dieses verdeutlichen. Die Zeit für den Austausch eines Datums ist dabei konstant T_k.

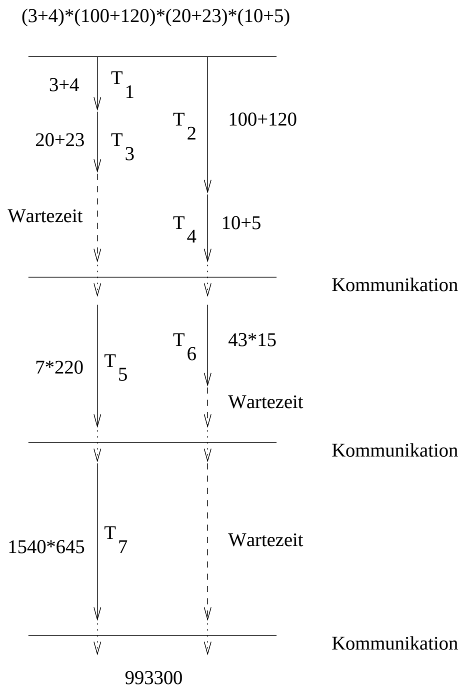


Abbildung 2.1 Beispiel für Kommunikationsverluste

  Es ergibt sich somit eine Kommunikationszeit von $T_K = 3 * T_k$ und eine Rechenzeit von $T_R = T_2 + T_4 + T_5 + T_7$. Für die Gesamtrechenzeit ist somit immer der längste Bearbeitungsstrang

[5] Auf das Problem gemeinsam genutzter Variablen wird im weiteren Verlauf noch näher eingegangen.

[6] In dem Beispiel wird davon ausgegangen, daß das Addieren und Multiplizieren großer Zahlen mehr Zeit kostet als das kleiner Zahlen. Dies ist der Fall, wenn man die Operationen „schulbuchmäßig“ durchführt, wie es z.B. in LISP passiert.

zwischen zwei Kommunikationsstellen maßgebend. Der *Speed-Up* und die Effizienz betragen damit

$$S(2) = \frac{\sum_{i=1}^{7} T_i}{T_2 + T_4 + T_5 + T_7 + 3 * T_k} \quad E(2) = \frac{\sum_{i=1}^{7} T_i}{2 * (T_2 + T_4 + T_5 + T_7 + 3 * T_k)}.$$

- ungleichmäßige Lastverteilung (load-balancing Problem)

  Das Beispiel in Abb. 2.1 zeigt, daß nicht immer alle Prozessoren gleichzeitig arbeiten. Es können durch Kommunikation Wartezeiten entstehen. Haben die Teilaufgaben der einzelnen Prozessoren einen unterschiedlichen Zeitbedarf, so verbringen die Prozessoren mit den kurzen Teilaufgabe einen Teil ihrer Zeit mit Warten. Dadurch sinkt die Effizienz. Bei der Entwicklung eines parallelen Algorithmus ist also besonders darauf zu achten, daß die parallellaufenden Teilaufgaben ungefähr den gleichen Zeitbedarf haben, oder daß immer genügend kurze Teilaufgaben vorhanden sind, die die Zeitdifferenz ausgleichen.

- Erzeugen der Threads und Aufsammeln der Ergebnisse

  Das Erzeugen von Threads kostet Zeit. Es müssen Verwaltungsstrukturen aufgebaut und die Threads auf die Prozessoren verteilt werden. Außerdem müssen evtl. die Endergebnisse aufgesammelt und dem Benutzer an einer definierten Schnittstelle übergeben werden.

Diese Einschränkungen zeigen, daß nicht jedes parallel zu lösende Problem auch sinnvoll zu parallelisieren ist. Der durch die Parallelität hervorgerufene Rechenaufwand muß sehr viel geringer sein als der eigentliche Rechenanteil, um einen Gewinn aus der Parallelisierung zu erzielen. Hierbei spielt die Granularität der Parallelisierung eine entscheidende Rolle. Je größer z.B. der Aufwand für Datenaustausch ist und je häufiger er stattfindet, desto größer müssen auch die einzelnen Teilprogramme sein. Bei einer Realisierung der Rechenoperation des Beispiels in Abbildung 2.1 als Maschinensprachebefehle wäre der Aufwand für die beschriebene Parallelisierung sicher zu hoch, wenn diese nicht auch auf der Maschinenebene stattfindet. Leider läßt sich dieses Verhältnis nicht immer im voraus abschätzen, so daß häufig erst eine Testimplementation zeigt, ob der Ansatz lohnenswert ist.

2.2 Hardware

Parallelrechner sind auf sehr unterschiedliche Weise realisierbar. Eine Übersicht über derzeit aktuelle Architekturen würde daher den Rahmen dieser Arbeit sprengen. Eine Klassifizierung der möglichen Architekturen hat Flynn in [Fly66] vorgenommen. Die Einteilung ist nicht optimal, da vor allem neuere Prozessorentwicklungen nicht eindeutig einer Klasse zugeordnet werden können. Die auf der algorithmischen Ebene beschriebenen Parallelisierungen spiegeln jedoch häufig die Klasse der Zielarchitektur wider. Flynn unterscheidet die Architekturen aufgrung ihrer unterschiedlichen Eingabeströme. Ein Strom ist dabei eine Sequenz aus Instruktionen oder Daten.

- SISD single instruction, single data stream

  Hier wird zu einem Zeitpunkt nur ein Befehlsstrom und ein Datenstrom abgearbeitet. Dieses entspricht der herkömmlichen von-Neumann-Architektur mit einem sequentiellen Programmfluß.

- SIMD single instruction, multiple data stream

  Bei dieser Klasse wird ebenfalls nur ein Befehlstrom abgearbeitet. Jeder Befehl kann jedoch parallel mit Daten aus unterschiedlchen Datenströmen ausgeführt werden. Die bekanntesten

Rechner dieser Art sind Vektorrechner, die sich besonders gut für die Lösung numerischer Probleme eignen (z.B. Matrixmultiplikation). Mit ihnen können jedoch nur Probleme parallel gelöst werden, deren einzelnen Teilaufgaben die gleiche Befehlsfolge haben. Man spricht dann von einer Vektorisierung des Problems.

- MISD multiple instruction, single data stream

  Die Klasse wird im allgemeinen als leer angesehen. Denkbar sind vielleicht Architekturen, bei denen sich die Prozessoren aus Sicherheitsgründen gegenseitig kontrollieren. Eine Realisierung dieses Konzepts ist aber nicht bekannt und wird von vielen Entwicklern als undurchführbar angesehen.

- MIMD multiple instruction, multiple data stream

  Diese Klasse ist die flexibelste. Mit ihr sind beliebige Parallelisierungsstrategien denkbar. Es existieren mehrere Befehls- und Datenströme. Jeder Befehlsstrom wirkt auf seinen eigenen Datenstrom. Unter diesen Begriff fallen sowohl die Multiprozessorrechner als auch der Zusammenschluß mehrerer *Workstations* zu einem virtuellen Parallelrechner, sog. **Workstationcluster**. Diese Cluster haben den Vorteil, daß sie ohne Mehrkosten zur Verfügung stehen, sobald die Workstations über ein Netz verbunden sind. Ihr Nachteil ist, daß sie über keinerlei spezielle Hardware für Parallelberechnungen verfügen und daher der Kommunikationsaufwand sehr hoch ist. Die einzelnen Teilprobleme müssen einen sehr hohen Rechenaufwand erfordern, damit sich diese Parallelisierungsart lohnt.

Eine näher an der Hardware ausgerichtete Klassifikation ist von Händler [Wal95] vorgeschlagen worden. Sie beschreibt die Parallelrechner qualitativ durch die Ebene auf der die Parallelität erzeugt wird und quantitative durch den Grad der Parallelität.

2.3 Programmiermodelle

Ein wesentlicher Faktor der parallelen Programmierung ist der Datenaustausch zwischen den Threads. Die beiden grundlegenden Modelle sind dabei **Message-Passing** und **Shared-Memory**. Obwohl sich die beiden Modelle theoretisch ineinander überführen lassen, bestimmt doch die jeweils verwendete Hardware, welches Modell als Grundlage einer Programmentwicklung dient.

2.3.1 Message-Passing

Im *Message-Passing* Modell verfügt jeder Thread nur über lokalen Speicher. Daten, die von mehreren Threads genutzt werden, werden mit Hilfe von Nachrichten ausgetauscht. Das Medium über das die Nachrichten übertragen werden, ist der Kanal. Jeder Thread, der an einem Kanal teilhat, kann über ihn mit den Befehlen *send(message,channel)* und *receive(message,channel)* senden und empfangen. Die Art der Kommunikation läßt sich nach der Anzahl und Art der Teilnehmer klassifizieren:

- 1:1 Auf dem Kanal existiert genau ein Empfänger und ein Sender.
- 1:n Ein Empfänger bekommt Nachrichten von mehreren Sendern.
- n:1 Mehrere Threads empfangen von einem Sender. Dabei sind zwei Fälle zu unterscheiden:

  - Eine einzelne Nachricht wird nur einem Thread zugestellt. Welcher Empfänger die Nachricht erhält, ist nicht vorher festgelegt.
  - Eine einzelne Nachricht wird allen Threads zugestellt. Dieses bezeichnte man als **broadcast**.

- n:m Mehrere Sender und Empfänger kommunizieren über einen Kanal. Auch hierbei kann man noch unterteilen in Nachrichten für alle, Nachrichten für einen einzelnen und Nachrichten für eine ausgewählte Gruppe von Threads.

Eine explizite Synchronisation zwischen den Threads ist beim *Message-Passing* nicht notwendig. Diese Aufgabe wird implizit durch die Art des Kanals erledigt. Zwei verschiedene Arten sind denkbar.

Asynchrones Message-Passing bedeutet, daß ein Sender unabhängig vom Empfänger senden kann, d.h. er braucht nicht zu warten, bis der Empfänger empfangsbereit ist. Programmtechnisch ausgedrückt, blockiert der *send()*-Aufruf den Sender nicht. Er kann sofort weiterarbeiten, ohne jedoch die Gewißheit zu haben, daß seine Nachricht auch beim Empfänger ankommt. Die abgeschickte Nachricht wird in einem theoretisch unendlich großen Puffer zwischengespeichert, bis der Empfänger bereit ist, sie entgegenzunehmen. Führt ein Empfänger ein *receive()* aus, ohne daß eine Nachricht im Puffer steht, wartet das *receive()* solange, bis der Puffer gefüllt wird. Dieses Verfahren weist damit keine Wartezeiten auf, solange der Puffer schnell genug gefüllt wird. Der Puffer entkoppelt die Threads, so daß keine Synchronisation zwischen ihnen notwendig ist. Damit ist die größtmögliche Unabhängigkeit der Threads voneinander gewährleistet. Nachteilig ist aber die Forderung nach einem unendlich großen Puffer, der nicht zu verwirklichen ist. Daher kann es auch bei einem *send()*-Aufruf zu einer Blockade kommen, wenn im Puffer kein Platz mehr ist. Außerdem müßte ein aufwendiges Acknowledge-Protokoll benutzt werden, falls der Sender sicher sein will, daß der Empfänger seine Nachricht erhalten hat.

Beim **synchronen Message-Passing** müssen beide Partner zur Kommunikation bereit sein. Der Initiator muß also solange warten, bis der entsprechende Partner bereit ist, die Nachricht auszutauschen.[7]. Jede Kommunikation beinhaltet also gleichzeitig eine Synchronisation, die Nachricht braucht nicht zwischengespeichert zu werden. Der Sender kann daher sicher sein, daß seine Nachricht ankommt. Ein Nachteil ist, daß es bei jedem Verschicken von Nachrichten zu Wartezeiten kommt.

2.3.2 Shared-Memory

In einem *Shared-Memory* System ist es den Threads möglich, neben ihrem lokalen Speicher auch auf einen gemeinsamen Speicherbereich (und damit gemeinsame Variablen)[8] zuzugreifen. Um Interferenzen der Threads beim Zugriff auf diese Variablen zu vermeiden, muß eine Synchronisation erfolgen. Das folgende Beispiel zeigt, welche Auswirkungen ein unsynchronisierter Zugriff haben kann. Dabei wird angenommen, daß jede arithmetische Operation sowie die Zuweisung an eine Variable atomar ist.

[7] Man nennt dies auch Rendezvous.

[8] Zu den sich daraus ergebenden Konsequenzen für die Hardware, wie Busarbitrierung- und *Cache-Coherency* Protokolle sei z.B. auf [SUN92] verwiesen.

```
void inc(int *var)
{
  *var=*var+1;
}

void main()
{
  int x,y;

  x=4;
  y=8;
  par
  {
    inc(&x);
    inc(&x);
    inc(&y);
   }
  print(x,y);
}
```

Mögliche Ausgaben:
x=5 y=9
x=6 y=9

Das Verhalten des Algorithmus ist also nicht deterministisch. Das Ergebnis hängt von der unvorhersehbaren Reihenfolge der Ausführung der einzelnen Befehle ab.[9] Eine sehr wichtige Rolle bei diesem Modell spielt die Synchronisation, die explizit vom Programmierer vorgenommen werden muß. Aufgrund der Bedeutung der Synchronisation sei hier kurz auf ihre Arten und Probleme eingegangen.

Grundsätzlich gibt es zwei Ansätze, um die Konsistenz von Daten zu gewährleisten. Im ersten Fall wird der Programmcode, der mit der Variable arbeitet, mit Hilfe einer **mutual-exclusion** (kurz *mutex*) Sperre[10] geschützt, so daß nicht zwei Threads gleichzeitig diesen Codeabschnitt ausführen können (**Codelocking**). Der entsprechende Abschnitt – in obigem Beispiel der Zugriff auf den Parameter *var* – wird dann als **critical section** bezeichnet. Die Prozedur *inc()* müßte daher wie folgt definiert werden:

[9] Man spricht bei einem solchen Fehler auch von einer *race condition*.

[10] Diese Sperre wird auch binärer Semaphor genannt und geht auf das Semaphorkonzept von Dijkstra [Dij68] zurück.

```
mutex_t lock_for_inc;

void inc(int *var)
{
  mutex_lock(lock_for_inc);
  *var=*var+1;
  mutex_unlock(lock_for_inc);
}
```

Die mögliche Parallelität der Anweisungen *inc(x)* und *inc(y)* würde damit aufgehoben. Im Gegensatz dazu steht das **Datalocking**. Dabei wird die Variable gesperrt, und nicht der Code, der die Variable benutzt. Für das Beispiel bedeutet das, daß mehrere Threads die Prozedur *inc()* gleichzeitig ausführen können, sofern der Parameter *var* an verschiedene Variablen gebunden ist. Das Programm sieht dann folgendermaßen aus:

```
  struct lock_int{ int value; mutex_t lock; };

void inc (int *var)
{
  *var:=*var+1;
}

void main()
{
  struct lock_int x,y;

  x.value=4;  /* keine Sperre notwendig, da kein konkurriender Zugriff
  möglich */
  y.value=8;
  par
  {
    /* Die Befehle innerhalb eines Blockes gelten als nicht
       parallelisierbar */
    {
      mutex_lock(x.lock);
      inc(&x);                /* Diese Variable ist jetzt geschützt */
      mutex_unlock(x.lock);
    }
    {
      mutex_lock(x.lock);
      inc(&x);
      mutex_unlock(x.lock);
    }
    {
      mutex_lock(y.lock);  /* Dieser Block kann echt parallel zu einem
```

Fortsetzung auf der nächsten Seite

```
                        der obigen ausgeführt werden.*/
      inc(&y);
      mutex_unlock(y.lock);
    }
  }
  print(x);
}
```

Bei dieser Art des Schutzes sind auch andere, feinkörnigere Schutzmechanismen realisierbar. Variablen, die wesentlich häufiger gelesen als geschrieben werden, können mit einer sog. Leser/Schreiber-Sperre versehen werden, d.h. mehrere Leser können gleichzeitig die Variable lesen, solange kein Schreiber sie verändern möchte. Synchronisationen führen aufgrund der vielen möglichen Zustände eines parallelen Systems sehr leicht zu Fehlern.[11]

Ein häufig vorkommendes Problem bei der Verwendung von Sperren ist der **Deadlock**, bei dem eine Menge von Threads vergeblich auf die Freigabe benötigter Ressourcen wartet und somit keinen Fortschritt mehr erzielt. Da die Reihenfolge, in der Sperren akquiriert werden, nicht deterministisch ist, kann ein Programm lange Zeit fehlerfrei laufen, bevor eine *Deadlock*-Situation auftritt. Dieses erschwert die Fehleranalyse beträchtlich. In der Regel wird ein Deadlock durch ein zyklisches Warten von Threads auf Sperren, die andere Threads halten, erzeugt. Ein Beispiel soll diesen Zusammenhang erläutern: Thread A und Thread B benötigen sowohl Sperre a als auch Sperre b. A besitze schon a und B Sperre b. A blockiert bei dem Versuch, Sperre b zu bekommen. B blockiert ebenfalls, wenn er Sperre a anfordert. Da keiner der beiden seine schon akquirierte Sperre wieder freigibt, warten beide unendlich lange. Eine vielbenutzte Strategie zur Vermeidung dieser *Deadlocks* ist die Definition einer Hierarchie auf den verwendeten Sperren. Ein Thread darf dann nur eine Sperre belegen, wenn er noch keine Sperre besitzt, die in der Hierarchie tiefer als die geforderte steht. Ist in dem Beispiel Sperre a größer als b, so könnte die beschriebene Situation nicht auftreten. Thread B hätte die Sperren in der beschriebenen Reihenfolge nämlich gar nicht anfordern dürfen. Die Verhinderung und Erkennung von *Deadlocks* ist ein großes Forschungsgebiet im Bereich der Betriebssysteme. Für eine ausführlichere Erklärung sei auf [Hof91] und [Dei84] verwiesen.

Der genau entgegengesetzte Fehler ist der **Livelock**. In diesem Fall sind die Threads so sehr mit dem Datenaustausch beschäftigt, daß sie der Problemlösung nicht näher kommen. Der *Livelock* ist eine Variante der Endlosschleife eines sequentiellen Programms bei der Parallelisierung.

2.4 Einführung in die SOLARIS Thread-Architektur

Das SOLARIS Betriebssystem von SUN ist ein multiprozessorfähiges Unix-Derivat. Wichtige Ziele bei der Implementation des *Multithreading*-Interfaces waren dabei die Unabhängigkeit von der zugrundeliegenden Hardware, eine kostengünstige Realisierung der Verwaltung und Synchronisation von Threads und eine Universalität, die es erlaubt, parallele Algorithmen mit unterschiedlichsten Anforderungen zu realisieren. Um diese Ziele zu erreichen, wurde von der ursprünglichen Auffassung eines Prozesses als ein Programmfluß Abstand genommen. Ein Prozeß kann mehrere parallellau-

[11] Die Probleme beim *Message-Passing* sind analog. Nur werden sie dort durch die implizite Synchronisation des Kommunikationssystems hervorgerufen.

fende Programmflüsse (=Threads of Control oder kurz Threads) haben. Jeder Thread kann dabei die gesamten Prozeßressourcen wie Datensegmente, Codesegmente, Filehandles etc. benutzen. Nur die Prozessorregister und der Stack sind für jeden Thread separat vorhanden, um eine unabhängige Ausführung zu gewährleisten. Die Threads stellen die Abstraktionsebene dar, auf der Programmentwickler parallelisieren. Sie erlaubt es, immer dann Nebenläufigkeit zu definieren, wenn es die Problemlösung zuläßt. Unabhängig arbeitende Teilprogramme müssen nicht in einen sequentiellen Ablauf gepreßt werden. Ein gutes Beispiel hierfür sind Applikationen mit einer graphischen Oberfläche, bei denen die einzelnen Fenster als unabhängige Teilprogramme gesehen werden können. Die Aufteilung des Problems in Threads spiegelt also die logische Parallelität wider. Dem Programmierer bleibt dabei verborgen, ob das Programm echt parallel auf einem Mehrprozessorsystem abläuft, oder ob die Parallelität nur simuliert wird. Bei manchen Problemen macht eine Parallelisierung aber nur Sinn, wenn die Teilprogramme echt parallel laufen, da sonst der Overhead für die Threadwechsel zu groß wird. Daher bietet SOLARIS eine weitere Abstraktionsstufe an: die **Lightweight Processes** (LWPs)[12]. Diese lassen sich als virtuelle Prozessoren auffassen. Sie sind die Ausführungseinheiten für die Threads. Die LWPs haben außerdem bestimmte Ressourcen wie virtuelle Zeit oder

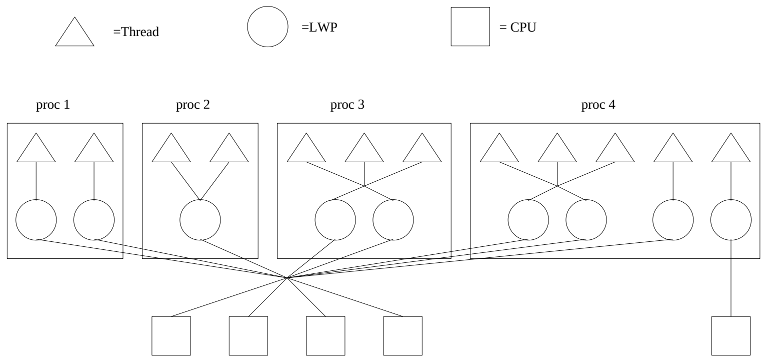


Abbildung 2.2 Abstraktionsebenen der SOLARIS Architekur (Entnommen [PKB+91])

alternative Signal-Stacks, die sie nicht an die Threads weitergeben. Sind diese Ressourcen für die Anwendung wichtig, gibt es die Möglichkeit, einen Thread fest an einen LWP zu binden. In diesem Fall führt der LWP nur diesen einen Thread aus. LWPs sind dem Betriebssystem bekannt, sie werden auf die vorhandenen Prozessoren abgebildet. Sind mehr LWPs als Prozessoren vorhanden sind, erfolgt durch eine Zuordnungsstrategie eine gleichmäßige Aufteilung der Prozessorleistung auf alle LWPs. Threads hingegen sind reine Benutzerstrukturen. Das Erzeugen eines Threads und die Synchronisation innerhalb eines Prozesses bedarf daher keinerlei Systemaufrufe. Dieses reduziert den Aufwand für die Parallelverarbeitung deutlich.[13] Darüber hinaus lassen sich Ressourcen eines Prozesses sehr einfach nutzen, da alle Threads eines Prozesses im gleichen Adressraum liegen. Öffnet z.B. ein Thread eine Datei, so können danach alle anderen Threads aus dieser Datei lesen, vorausgesetzt, der Dateideskriptor liegt innerhalb ihres syntaktischen Zugriffsbereiches. Probleme bereitet der gemeinsame Adressraum bei der Benutzung threadspezifischer statischer Daten. Daten sind in herkömmlichen Programmiersprachen normalerweise global oder lokal. Globale Daten werden auf eine bestimmte Speicheradresse eines Datensegments abgebildet, lokale Daten auf dem Stack verwaltet. Lokale Daten sind damit automatisch threadspezifisch, da jeder Thread seinen eigenen Stack hat. Eine threadspezifische globale Variable benötigt dagegen eine Speicheradresse

[12] Die Definition von LWPs in SOLARIS ist weder kompatibel mit der des SUN-OS 4.X noch anderer in der Literatur verwendeter Definitionen.

[13] zu Performanzmessungen siehe SUN White Papers [PKB+91] und Anhang B

pro Thread, auf die aber möglichst transparent zugegriffen werden sollte. SOLARIS stellt dazu das **Thread-Specific-Data (TSD)-Interface** zur Verfügung. Für jede TSD-Variable wird dazu ein Zugriffschlüssel erzeugt, der dann zum Speichern und Lesen der Variable benutzt wird. Dadurch wird keinerlei Unterstützung des Compilers für TSD-Variablen benötigt. TSD-Variablen spielen vor allem bei der Parallelisierung von bestehenden sequentiellen Programmen eine große Rolle. Da dieses auch Gegenstand der vorliegenden Arbeit ist, wird hier genauer auf ihre Verwendung eingegangen.

2.4.1 Threadspezifische Daten in SOLARIS

Auf eine TSD-Variable wird unter SOLARIS mit Hilfe der beiden Funktionen *thr_setspecific()* und *thr_getspecific()* zugegriffen. Die Realisierung von TSD-Variablen über Funktionen hat einen entscheidenden Nachteil. Ein ursprünglich einfacher Speicherzugriff wird in einen Funktionsaufruf umgewandelt. Dadurch verlängert sich die Laufzeit eines Programms erheblich, falls dieses TSD-Variablen häufig benutzt. Die Definition der Zugriffsfunktionen erschwert die Verwendung von TSD-Variablen zusätzlich. Die Argumentstruktur ist wie folgt aufgebaut:

*int thr_setspecific(key_t key, void *value)*
*int thr_getspecific(key_t key, void **valuep)*

Key ist dabei der zuvor erzeugte Zugriffsschlüssel. *Thr_setspecific()* bindet *value* an *key*, *thr_getspecific()* speichert den aktuell gebundenen Wert von *key* an der Adresse, auf die *valuep* zeigt, ab. Dadurch, daß die Adresse nicht als Ergebnis der Funktion zurückgegeben wird, sind sogar zwei Funktionsaufrufe notwendig, wenn man für das gesamte Programm transparent eine globale Variable in eine TSD-Variable umwandeln will. Zur Verdeutlichung sei der Vorschlag von SUN zur transparenten Umsetzung angeführt.[SUN94][14]

```
 /* mywindow sei in dem sequentiellen Programm eine globale Variable
 gewesen, die den Handle für einen Ausgabestream enthielt */

/* ersetzte alle Referenzen auf mywindow durch Aufrufe der Funktion
_mywindow() */
#define mywindow _mywindow()

thread_key_t mywindow_key;  /* Der Zugriffschlüssel für mywindow */
mutex_t lock;               /* Mutal exclusion für Schlüsselerzeugung */
FILE *_mywindow(void) {
   FILE *win;

  thr_getspecific(mywindow_key,&win);
  return(win);}
```

Fortsetzung auf der nächsten Seite

[14]Das Original enthält einige Fehler, daher wird hier eine korrigierte Version abgedruckt. Sie ist aber nicht getestet und soll nur das Problem verdeutlichen.

```
/* Erzeugt je Thread einen Speicherbreich für mywindow, damit jeder
 Thread in sein eigenes Fenster schreiben kann. */
void make_mywindow(void){
 FILE **win;
 static int once=0;

 mutex_lock(&lock);
 if (!once)        /* Der Schlüssel darf nur einmal erzeugt werden */
  {
    once = 1;
    thr_keycreate(&mywindow_key, free_key);
  }
  mutex_unlock(&lock);
  win=malloc(sizeof(FILE*)); /* Speicherplatz für die TSD-Variable */
  create_window(win,...);     /* Erzeugen des Fensters pro Thread */
  thr_setspecific(mywindow_key,win);}

/* Freigabefunktion; diese wird beim Beenden des Threads aufgerufen */
void free_key(void *win); {
 free(win);}

/* Diese Funktion wird als erste nach Erzeugen eines Threads aufgerufen */
void start_thread(...) {
 ...
 make_mywindow();
 ...}
```

Implementiert man z.B. den Stack des parallelen LISP, der bei fast jedem Funktionsaufruf benutzt wird, in dieser Art, so ergibt sich eine Verfünffachung der Laufzeit bei dem vorliegenden Parser.[15] Um den Overhead durch die Parallelisierung nicht zu groß werden zu lassen, mußten die häufig benutzten TSD-Variablen also auf eine andere Art realisiert werden.

2.4.2 Realisierung threadspezifischer Daten über Register

Für eine alternative Realisierung von TSD-Variablen ist die SPARC-Architektur sehr hilfreich. Der SPARC-Prozessor stellt nämlich mehrere globale Register zur Verfügung, die von keiner Bibliotheksroutine benutzt werden sollten, und somit der Anwendung zur Verfügung stehen. Diese Register sind aber threadspezifisch, so daß sie direkt als Zeiger auf eine TSD-Variable genutzt werden könnten.[16] Die Zahl der im Programm verwendeten TSD-Variablen übersteigt im allgemeinen jedoch die Zahl der freien Register. Daher wird eine Indirektionsstufe eingeführt. Ein Register enthält nicht mehr direkt den Zeiger auf eine TSD-Variable, sondern einen Zeiger auf ein Feld, dessen n-tes Element den Zeiger zur TSD-Variablen enthält (siehe Abb. 2.3).

[15] Bei einem Testlauf des Parsers wurde die Funktion *thr_getspecific()* mehr als 86 Millionen mal angesprochen.

[16] Ist die Variable von einem Typ, der nur ein Wort benötigt, so könnte man diese Variable sogar direkt im Register ablegen.

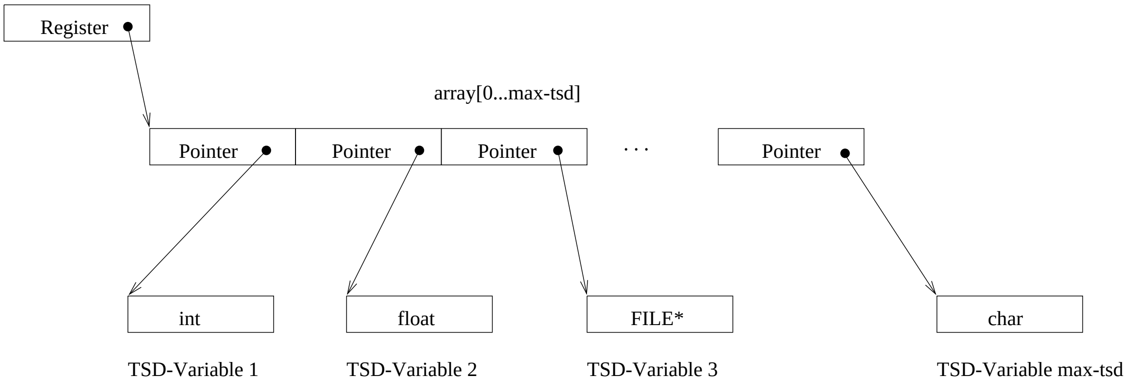


Abbildung 2.3 Adressierung von TSD-Variablen über Register

Dabei wird schon im Quellcode festgelegt, in welchem Feldelement einen spezielle TSD-Variable steht. Der doppelte Funktionsaufruf wird damit durch eine doppelte Zeigerdereferenzierung ersetzt. Durch diese Änderung vergrößerte sich die Laufzeit des Parsers mit der *multithreading*-fähigen Bibliothek nur um 15 Prozent. Dynamische TSD-Variablen müssen weiterhin mit dem SOLARIS TSD-Interface erzeugt werden. Der Vorteil der höheren Geschwindigkeit wird natürlich durch eine massive Hardwareabhängigkeit erkauft. Ein weiteres Problem in der Praxis ist das Ansprechen eines Registers aus einem C-Programm heraus. Dieses ist zwar bei vielen C-Compilern möglich, jedoch ist es kein Bestandteil der ANSI-Norm, so daß jeder Compiler seine eigene Erweiterung anbietet. Ein weiterer Punkt erweist sich als besonders schwerwiegend, selbst wenn ein SPARC-Prozessor mit passendem C-Compiler zur Verfügung steht. Die Idee, globale Register ausschließlich dem Anwender zur Verfügung zu stellen, erfordert aber die Kooperation aller Komponenten eines Programms. In der Praxis ist man darauf angewiesen, daß sich die Entwickler von Bibliotheksroutinen ebenso an dieser Forderung halten, wie die Compiler. Dieses tut jedoch nicht einmal SUN selbst. Im Laufe der Arbeit stellte sich heraus, daß alle blockierenden Threadbibliotheksaufrufe die Inhalte der globalen Register verändern.[17] Vor jedem Aufruf einer Funktion, die blockieren kann, müssen also die benutzten Register gesichert und nach Beendigung wieder hergestellt werden. Dieser zeitraubende Mechanismus ließ sich bei der Implementation jedoch umgehen, weil die SOLARIS-Sourcen zur Verfügung standen. Damit konnte die Threadbibliothek entsprechend geändert werden. Erst diese Modifikation ließ es zu, eine effiziente Parallelversion des verwendeten LISP zu implementieren.

[17] Dieses betrifft nicht nur die eigentlichen Threadroutinen und Synchronisationsfunktionen, sondern auch alle Bibliotheksfunktionen, die diese intern benutzen, wie z.B. *malloc()* oder *printf()*.

3
ASPEKTE EINES PARALLELEN LISP

Der vor Beginn der Arbeit vorhandene sequentielle Parser ist in LISP implementiert. Die Grundlage der parallelen Version des Parsers muß daher ein paralleles LISP sein. Da das verwendete LISP (Allegro Common LISP, kurz ACL) nicht im Quellcode vorliegt, kann dieses nicht zu einem parallelen LISP für ein Mehrprozessorsystem erweitert werden. Der daher eingeschlagene Weg scheint auf den ersten Blick vielleicht etwas ungewöhlich, bietet jedoch auch Vorteile gegenüber einer Weiterentwicklung eines im Quellcode vorhandenen LISP. Der Parser wird zunächst mit Hilfe eine Übersetzers (LISP-to-C Translator) von LISP in ein C-Programm umgewandelt und kann dann mit einem beliebigen C-Compiler kompiliert werden. Der Übersetzer erzeugt dabei lesbare C-Programme, d.h. Bezeichner und Programmkonstrukte werden soweit wie möglich übernommen. Dieses erlaubt eine Bearbeitung des Parsers auf der C-Ebene. Außerdem wird so die Parallelisierung durch entsprechende Hilfsprogramme des Betriebssystems wie Profiler und Debugger erleichtert. Durch direktes Einfügen von C-Funktionen ist es möglich, die Threadschnittstelle direkt aus LISP-Programmen anzusprechen. Es ist nicht notwendig, die gesamte Threadschnittstelle auf die LISP-Ebene zu ziehen. Erforderlich ist jedoch eine Anpassung der Übersetzer-Bibliothek, die zu jedem C-Programm benötigt wird. Diese bildet im Prinzip die Funktionalität von LISP in C nach. Einige besonders interessante Aspekte dieser Anpassung, die im folgenden immer als LISP-Parallelisierung bezeichnet wird, stellt dieses Kapitel vor. Es handelt sich dabei um Probleme, die jede parallele LISP-Implementation zu lösen hat.

3.1 Ziele der Implementation

Die Parallelisierung einer Programmiersprache kann mit zwei sehr unterschiedlichen Zielen verfolgt werden. Zum einen kann man versuchen, unabhängige Teilprogramme automatisch erkennen zu lassen. Dazu müßten Compiler und Laufzeitsystem so erweitert werden, daß sie parallel ausführbare Programmteile erkennen und diese automatisch umsetzen. Bestehende Programme könnten dann unverändert übernommen werden. Einen Überblick der Ansätze dieser Art für LISP gibt [Hal89, S. 71]. Inwieweit mögliche Parallelität automatisch erkannt werden kann, ist aber fraglich. Automatische Parallelisierer müßten die Programme „verstehen“, die sie parallelisieren. Eingesetzt werden sie daher fast ausschließlich bei numerischen Programmen, die sehr viele Schleifenkonstrukte und Feldberechnungen beinhalten und damit leicht zu parallelisieren sind. Die meisten Erweiterungen von Programmiersprachen gehen einen anderen Weg. Die Aufteilung des Programms wird dem Entwickler überlassen, die Programmiersprache stellt nur passende Konstrukte zur Verfügung.

Genau dieses Ziel besitzt auch die vorliegende Implementierung. Dabei sind diese Konstrukte eine Abstraktion der *Multiprocessing*-Fähigkeit von ACL[1] und der SOLARIS-Threadarchitektur. Programme können unter ACL entwickelt und getestet werden. Erst für eine echt parallele Ausführung der Threads muß das Programm nach C übersetzt und kompiliert werden (Abb. 3.1).

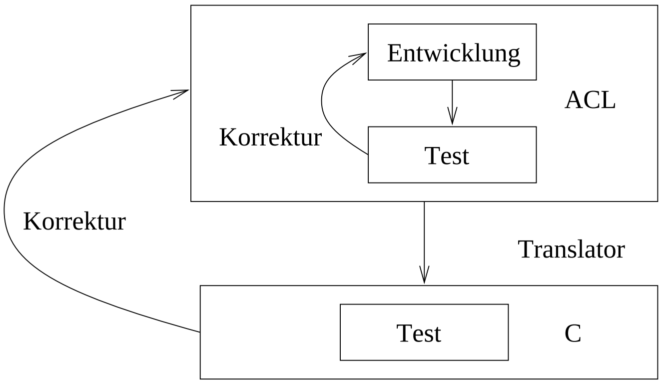


Abbildung 3.1 Ablaufdiagramm der Entwicklung eines Multithreading-LISP-Programms

Als minimale Anforderung muß es daher eine Funktion zur Erzeugung von Threads geben (*(new-thread ...)*), sowie Möglichkeiten, auf die Terminierung von Threads zu warten (*(wait-for-threads ...)*). Des weiteren ist die Synchronisation bei Zugriff auf globale Ressourcen zu unterstützen (*(get-lock ...)* und *(release-lock ...)*). Dazu wird der Datentyp *lock* eingeführt. Als Erweiterung sind Funktionen für die effizente Implementierung von Zustandsänderungen (Conditions) realisiert (*(wait-for-condition ...) und (change-condition ...)*). Die komplette LISP-Bibliothek des Übersetzers muß somit *multithreading*-fähig sein und um die genannten Funktionen erweitert werden. Auf einige problematische Aspekte der Parallelisierung wird in den weiteren Abschnitten genauer eingegangen.

3.2 Speicherverwaltung

Eine der besonderen Eigenschaften von LISP ist die automatische Speicherverwaltung. Der Programmierer braucht sich nicht um die Allokation und Freigabe des Speichers für seine Daten zu kümmern. Jedesmal wenn ein LISP-Programm Speicher für Daten braucht, sorgt das LISP-System für einen passenden Speicherbereich. Die Freigabe des Speichers übernimmt der sog. **garbage collector** (GC). Ein Speicherbereich kann immer dann freigegeben werden, wenn das LISP-Programm auf diesen Bereich keinen Zugriff mehr hat. Dieses sei an folgendem Beispiel verdeutlicht:

```
(setf a (list 1 2 3))
(setf a (list 3 2 1))
```

Nach dieser Befehlsfolge kann das Programm die Liste *'(1 2 3)* nicht mehr ansprechen. Sie ist unbrauchbar (*garbage*) geworden. Spätestens wenn kein freier Speicher mehr für neue Objekte

[1] ACL stellt eine *Multiprocessing-Package* zur Verfügung. Diese erlaubt eine logische Parallelisierung, die jedoch auch bei vorhandenem Mehrprozessorsystem nicht in echte Nebenläufigkeit umgesetzt wird.

vorhanden ist, wird der Speicherplatz für diese *garbage*-Objekte wiedergewonnen. Diese Art der Speicherverwaltung stellt natürlich hohe Anforderungen an die Entwicklung eines LISP-Systems. Wenn der GC aktiv wird, muß auf jeden Fall sichergestellt werden, daß alle noch benötigten Objekte die Speicherfreigabe „überleben". Diese sind z.B. statisch allozierte, globale Variablen, lokale Variablen und Parameter der gerade aufgerufenen Funktionen. Andererseits muß der Speicherplatz jedes *garbage*-Objekts auch irgendwann einmal wieder freigegeben werden. Ein GC erfüllt also im wesentlichen zwei Funktionen:

- unbenutzte Objekte erkennen *(garbage detection)*
- Speicherplatz unbenutzter Objekte wiederverwenden *(garbage reclamation)*

Unbenutzte Objekte werden mit Hilfe eines Erreichbarkeitskriteriums erkannt. Dieses wird normalerweise mittels eines fest definierten Aufhängers, dem **root-set**, definiert. Alle Objekte, die von diesem *root-set* aus über beliebige Verweisketten (Pfade) erreichbar sind, gelten als benutzt, alle anderen als unbenutzt. Wiedergewonnen wird der Speicherplatz, den diese unbenutzten Objekte belegen, indem vermerkt wird, daß er für künftige Speicheranforderungen zur Verfügung steht. Drei GC-Verfahren (mark-and-sweep, stop-and-copy und inkrementelle GCs) sollen hier kurz vorgestellt und deren Parallelisierungsmöglichkeit erörtert werden.[2]

Mark-and-Sweep ist eine sehr geradlinige Implementierung der beiden GC-Funktionen. In einer ersten Phase werden alle Objekte, die das Erreichbarkeitskriterium erfüllen, markiert. In der zweiten Phase wird der gesamte Speicher nach unmarkierten Objekten durchsucht und diese dann in Freispeicherlisten eingehängt. Diese einfache Methode hat mehrere Nachteile. Erstens ist der Zeitbedarf des GCs proportional zur Größe des Speichers. Jedes erzeugte Objekt muß mindestens einmal aufgesucht werden. Des weiteren wird der Speicher durch Objekte variabler Größe fragmentiert. Benutzte Objekte bleiben immer an ihrer Speicheradresse. Zwischen den benutzten Objekten sind Lücken, die aus der Wiedergewinnung des Speichers unbenutzter Objekte entstanden sind. Es kann also geschehen, daß ein für ein neues, großes Objekt kein Speicher mehr zur Verfügung steht, weil keine passende Lücke vorhanden ist. Führt man für jede Objektgröße eine eigene Freispeicherliste und kombiniert kleine aneinandergrenzende Lücken wieder zu größeren, läßt sich das Problem reduzieren. Ein echte Lösung ist damit aber noch nicht gegeben. Der *Mark-and-Sweep Collector* wird daher vielfach um einen Kompaktifizierungsalgorithmus erweitert, der die benutzten Objekte zu einem kontinuierlichen Speicherbereich zusammenfaßt.

Der **Stop-and-Copy Collector** vermeidet diese Nachteile des *Mark-and-Sweep Collectors*. Ein *Stop-and-Copy Garbage-Collector* teilt den Speicher in zwei gleichgroße Teile (**semi-spaces**) auf. Speicher wird aber nur aus einem Teil alloziert. Dieses ist der aktive Teilbereich. Der andere wird während des normalen Programmablaufs nicht benutzt. Der *Stop-and-Copy Collector* benötigt bei gleichgroßem nutzbarem Speicher doppelt soviel Hauptspeicher wie der *Mark-and-Sweep Collector*. Systeme mit virtueller Speicherverwaltung lagern den nicht benötigten *semi-space* aber automatisch aus. Der wesentlich größere Speicherbedarf kostet somit keinen echten Hauptspeicher. Ist der aktive Speicherbereich (*from-space*) voll, so wird der GC aufgerufen. Dieser sucht alle erreichbaren Objekte und kopiert sie in den inaktiven Bereich (*to-space*). Sind alle Objekte kopiert, wird der inaktive Bereich der aktive. Damit sind alle *garbage*-Objekte automatisch wieder freigegeben. Jedes Objekt darf aber nur einmal kopiert werden, auch wenn es über mehrere Pfade aus dem *root-set* erreichbar ist. Trifft der GC auf ein Objekt im *from-space*, so muß er erkennen, wenn dieses schon kopiert ist. Dann ist nur den Verweis, der zu dem Objekt geführt hat, zu aktualisieren. Dazu wird

[2] Für eine gute Einführung in sequentielle GC-Verfahren sei auf [Wil92] verwiesen.

ein sog. **Forwarding-Pointer** eingesetzt. Wenn ein Objekt kopiert wird, hinterläßt der GC in dem alten Objekt ein Verweis auf die neue Position des Objekts im *to-space*. Um festzustellen, welche Objekte schon verschoben sind, ist also ein Prädikat notwendig, daß zu jedem Objekt angibt, ob es verschoben wurde.[3] Durch das Kopieren der Objekte ergibt sich also automatisch eine Kompaktifizierung des Speichers. Neue Objekte werden immer am Anfang des freien Speichers abgelegt. Die Zeit für eine *Garbage-Collectorzyklus* ist nur noch proportional zu der Anzahl der erreichbaren Objekte. Im ungünstigsten Fall, wenn zwischen zwei GC-Zyklen kein Objekt *garbage* wird, werden wie beim *Mark-and-Sweep Collector* alle Objekte aufgesucht. Man kann aber in LISP davon ausgehen, daß viele Objekte sehr kurzlebig sind. Sie werden erzeugt und wieder verworfen, bevor der GC aufgerufen wird. Damit haben diese Objekte keinen Einfluß auf die Laufzeit des GCs.

Inkrementelle GCs stellen einen ganz anderen Ansatz dar. Hierbei wird nicht gewartet, bis der gesamte Speicher erschöpft ist, bevor der GC aktiv wird. Das laufende Programm wird immer wieder kurz unterbrochen, um einen kleinen Teil des Speichers zu bearbeiten. Ein *Garbage-Collector* Zyklus über den kompletten Speicherbereich teilt sich damit in viele kleine Abschnitte auf, in denen abwechselnd das Programm und der GC laufen. Weil während eines GC-Durchlaufes mit den Objekten gearbeitet wird, ist für dieses Verfahren ein besonders aufwendiges Protokoll notwendig, um das Erreichbarkeitskriterium sicher zu bestimmen.

Die Parallelisierungsmöglichkeiten eines GC teilt Halstead in [Hal89] in drei Klassen ein:

- inkrementelle GCs
- dedizierte GCs[4]
- nicht-inkrementelle GCs

Inkrementelle GCs scheinen für eine Parallelisierung am besten geeignte zu sein, da ein verzahntes Ablaufen von GC und LISP-Programm schon im Algorithmus spezifiziert ist. Außerdem kann man durch den Wechsel vom Programm- in den GC-Modus die Wartezeit bei einer Synchronisation sinnvoll nutzen. Sie sind jedoch sehr aufwendig zu implementieren und bedürften einer Anpassung der gesamten Übersetzer-Bibliothek. Aufgrund eines sehr hohen Synchronisationsaufwands beim Zugriff auf Speicherobjekte ist außerdem eine Unterstützung durch die Hardware wüschenswert, um eine effiziente Verarbeitung zu gewährleisten.

Bei dediziertem GC wird ein Thread nur für das GC abgestellt. Auch dieser Ansatz stellt eine Art inkrementeller GCs dar, da hier ebenfalls Programm und GC ineinander verzahnt ablaufen. Die Argumente gegen eine Realisierung in dieser Art sind daher analog zu denen des inkrementellen GCs.

Der einfachste parallele arbeitende GC ist der nicht-inkrementelle GC. Wenn kein Speicher mehr frei ist, werden alle Threads in ihrer aktuellen Programmausführung gestoppt und ein GC-Zyklus eingeleitet. Dabei können entweder alle Threads am GC-Verfahren teilnehmen, oder die Arbeit wird von dem Initiator erledigt. Ein bestehendes System mit einem nicht-inkrementellen GC läßt sich so am einfachsten erweitern. Die Art des GC, ob z.B. *Mark-and-Sweep* oder *Stop-and-Copy*, spielt

[3] Für den Fall, daß die *semi-spaces* durch einen kontinuierlichen Hauptspeicherbereich dargestellt werden, kann dieses z.B. ein Adressvergleich der *semi-space*-Grenzen mit dem *Forwarding-Pointers* sein. Andernfalls ist evtl. ein zusätzlicher Eintrag bei jedem Objekt notwendig.

[4] Halstead spricht von „GC on the fly“ [Hal89, S. 88].

dabei eine untergeordnete Rolle. Daher wird auch in der beschriebenen Implementation auf dieses Verfahren zurückgegriffen. Diese Realisierung wird im folgenden genauer dargestellt.

Für LISP-Funktionen gibt es normalerweise nur einen Eintrittspunkt in die Speicherverwaltung, nämlich die Funktion, die Speicher für das zu erzeugende Objekt reserviert. Im parallelen Fall muß sichergestellt werden, daß ein und derselbe Speicherbereich nicht doppelt vergeben wird. Weil die vorliegende Implementierung mit einem globalen Speicherbereich für alle Threads arbeitet, ist die Allozierungsfunktion damit eine *critical section*. Nur ein Thread kann zu einem Zeitpunkt Speicher besorgen. Daher kann auch nur dieser Thread feststellen, ob noch genügend Platz vorhanden ist. Stellt er fest, daß ein GC notwendig ist, wird dieses den anderen Threads signalisiert. Sie können nicht einfach gestoppt werden, weil sie eventuell noch nicht alle benutzten Objekte wie Parameter oder lokale Variablen vor dem GC geschützt haben. Sobald ein Thread eine Stelle erreicht hat, an der aus seiner Sicht ein GC erfolgen kann, meldet er dieses dem Initiator des GC zurück und geht in einen Wartezustand über. Haben alle Threads ihre Bereitschaft zum GC gemeldet, kann der Initiator einen GC Zyklus ausführen. Danach signalisiert er allen anderen Threads, daß sie ihre Arbeit fortsetzen können. Die Entscheidung, an welchen Programmstellen eine Bereitschaft zum GC signalisiert werden kann, liegt im Ermessen der LISP-Implementierung. An einigen Stellen ist diese Bereitschaft jedoch zwingend notwendig. Natürlich rechnet eine LISP-Funktion mit einem GC, wenn sie Speicher anfordert. Aber auch an allen Stellen, an denen eine LISP-Funktion blockieren kann wie I/O-Operationen oder Synchronisation, ist eine Bereitschaft zum GC zwingend notwendig, weil es sonst zu einem *Deadlock* kommen kann. Um die Eingriffe in die Übersetzer-Bibliothek so gering wie möglich zu halten, wurden genau diese Programmstellen ausgewählt, um eine GC-Bereitschaft zu signalisieren.

Aus der Art der Realisierung sind einige Nachteile ersichtlich. Zum einen beinhaltet der exklusive Zugriff auf den Allozierer einen Engpaß, weil das Anfordern von Speicher ein sehr häufiger Vorgang in LISP ist. Zum anderen leidet die Effizienz des parallelen LISP durch die auftretenden Wartezeiten bei einem GC. Der erste Nachteil läßt sich durch eine Erweiterung, die der Übersetzer bietet, die sog. **Local-Area**, etwas abschwächen. Eine *Local-Area* ist ein weiterer Speicherbereich, der analog zum normalen LISP-Speicher verwaltet wird. Im Unterschied zum normalen Speicher wird ihr Inhalt jedoch nicht vom GC bearbeitet. Eine Freigabe aller Objekte erfolgt automatisch, nachdem die Funktion, die die Auswertung innerhalb der *Local-Area* angestoßen hat, beendet ist. Erhält jeder Thread eine *Local-Area*, so läßt sich der Synchronisationsaufwand erheblich reduzieren. Viele LISP-Programme erzeugen nämlich wesentlich mehr lokale kurzlebige Objekte als globale langlebige.[5] Die Verringerung der Wartezeiten bei einem GC läßt sich jedoch nicht so leicht erzielen. Hier müßte auf die vorher erörterten GC-Verfahren zurückgegriffen werden, deren Realisierung den zeitlichen Rahmen dieser Arbeit aber gesprengt hätte.

3.3 Behandlung von Special-Variables

Durch **Special-Variables** wird in LISP das Konzept des dynamischen Gültigkeitsbereichs von Variablen realisiert. Wird eine *Special-Variable* innerhalb einer Umgebung an einen Wert gebunden, so ist diese Bindung so lange aktiv, wie die Umgebung existiert. Danach wird der alte Wert restau-

[5] Natürlich ist dieses eine Abschwächung des Konzepts, daß sich der Programmierer nicht um die Speicherverwaltung zu kümmern braucht. Die Entscheidung, wann ein Objekt nicht mehr benötigt wird, liegt hiermit wieder beim Programmierer.

riert. Zwei Möglichkeiten, diese Art der Bindung zu implementieren, sind das **Deep-Binding** und das **Shallow-Binding**. Beim *Deep-Binding* werden alle Paare (Variable, Wert) auf einen Stack gelegt. Wird der Wert für eine bestimmte Variable gesucht, so wird das oberste Paar, das zu dieser Variable gehört, geholt. Je nach Tiefe dieses Stacks kann die Suche recht lange dauern. Das *Shallow-Binding* umgeht dieses Problem, indem es für jede *Special-Variable* einen globalen Speicherplatz anlegt. Dieser enthält immer den aktuellen Wert. Wird der aktuelle Wert durch eine neue Bindung verdeckt, so wird er genauso wie beim *Deep-Binding* auf einen Stack gelegt. Die Bindung wird rückgängig gemacht, indem der oberste zur Variable gehörende Wert vom Stack genommen und in den globalen Speicherplatz geschrieben wird. Die dadurch erzielte konstante Zugriffszeit wird also durch zusätzliche Operationen beim Erzeugen einer neuen Bindung sowie Verlassen einer Umgebung erkauft. Bild 3.2 verdeutlicht beide Verfahren.

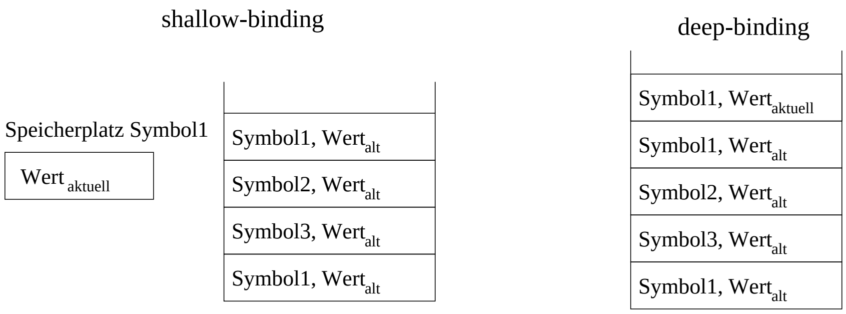


Abbildung 3.2 Realisierung von *Special-Variables*

In einem parallelen LISP ist die Semantik von *Special-Variables* zu klären. Sieht man die *Special-Variables* als prozeß-globale Variablen, so würde eine neue Bindung eines Thread von allen anderen sichtbar sein. Hierbei können die bekannten Zugriffskonflikte für globale Variablen auftreten, so daß dann eine Synchronisation durch den Programmierer erfolgen muß. Dabei wären keine Änderungen am LISP-System gegenüber einer sequentiellen Implementierung notwendig. Die gängigere Auffassung von *Special-Variables* in parallelen LISP-Systemen ist die einer threadspezifischen *Special-Variable*. Ein Thread erbt die Variablenbindungen seines Erzeugers. Neue Bindungen beeinflussen andere Threads aber nicht. Bei der Verwendung von *Deep-Binding* läßt sich diese Semantik sehr einfach implementieren. Jeder neue Thread kopiert einfach den Stack seines Erzeugers. Die Verwendung von *Shallow-Binding* erschwert die Parallelisierung etwas. Ein globaler Speicherplatz kann natürlich nicht mehrere Werte gleichzeitig annehmen. Für jeden Thread muß also eine eigene Eintragung vorhanden sein. Dieses entspricht der Umwandlung einer globalen in eine TSD-Variable. Der Bindungs-Stack ist ebenso wie beim *Deep-Binding* zu kopieren.[6] Allegro hat in seiner *Multiprocessing*-Erweiterung beide Semantiken realisiert. Generell sind *Special-Variables* threadspezifisch. Es gibt jedoch einen speziellen *setq* Befehl, der den Wert einer *Special-Variable* in allen Threads setzt (siehe [Fra92, Kap. 2.3.1]). An diesem Punkt unterscheiden sich die Übersetzer-Bibliothek und ACL. Die Übersetzer-Bibliothek kennt nur globale *Special-Variables*. Einzige Ausnahme ist die Variable ***worker-number***. Diese ist dem Übersetzer bekannt und wird gesondert als *TSD-Special-Variable* behandelt. Sie ist dazu gedacht, eine eindeutige fortlaufend numerierte Kennung für jeden Thread zu implementieren. Damit lassen sich dann pseudo-TSD Variablen implementieren. Dazu ein kleines Beispiel:

[6] Eine Diskussion über *Special-Variables* in einer parallelen LISP-Umgebung findet sich in [Jac90].

```
(defvar *worker-number*)
(defvar *my-own-string-array* (make-array 5))  ;; Maximal 5 Threads

(defmacro *my-own-string* ()
   (aref *my-own-string-array* *worker-number*)
)

(defun start-thread (id)
  (let ((*worker-number* id))
     (setf (*my-own-string*) (format nil "This is Thread ~A" id))
     (print (*my-own-string*))
  )
)
(dotimes (j 4)
  (new-thread 'start-thread j)
)
```

3.4 Catch-Throw Problematik

Die beiden Funktionen *catch* und *throw* bieten in LISP die Möglichkeit, sog. nicht-lokale Rücksprünge zu definieren. Die Form *(catch tag expression)* erzeugt einen *catcher* mit dem durch *tag* beschriebenen Marker[7] und wertet den Ausdruck *expression* aus. Wird während der Auswertung auf die Form *(throw tag value)* getroffen, wobei *tag* der passende Marker zum vorangegangenen *catch* ist, liefert *catch* den Wert *value* zurück. Andernfalls wird der Wert des Ausdrucks *expression* zurückgegeben. Solange *catch* und *throw* von ein und demselben Thread ausgeführt werden, ergeben sich auch in der parallelen Version keine Probleme. Wenn aber wie im folgende Programmbeispiel während der Auswertung von *expression* ein neuer Thread erzeugt wird, der dann auf ein passendes *throw* trifft, sind verschiedene Weiterführungen denkbar.

Erzeuger-Thread:

```
⋮

 (catch underflow
    (new-thread
       'calculate-something nil))

⋮
```

Erzeugter Thread (innerhalb der *calculate-something* Funktion):

```
⋮

  (setf x 4)
  (setf y 5)

⋮

  (if (< x y)
    (throw underflow -1)
  )
```

Zunächst könnte man sagen, jeder Thread reagiert nur auf einen *throw* aus seinem eigenen Programmfluß. Andererseits ist auch eine Semantik denkbar, in der der Thread, der auf einen *throw* trifft, dem Erzeugerthread dieses in irgendeiner Form (z.B. über Signale) mitteilt. Die vorliegende Implementation verschließt sich dieser Problematik, indem sie *catch/throw*-Konstrukte über

[7] Dieser Marker ist eine eindeutige Kennung.

Threads hinweg nicht zuläßt. Dieses liegt hauptsächlich an der Art, wie diese auf der C-Ebene realisiert sind. Jedes *catch/throw*-Paar wird in eine *setjump/longjump*-Kombination[8] umgewandelt. Das SOLARIS-Betriebssystem verbietet es aber, daß ein *longjump* eines Threads auf einen *setjump* eines anderen erfolgt. Diese Einschränkung betrifft aber nicht nur den Entwickler, sondern auch die Übersetzer-Bibliothek. Exception-Handler wie z.B. der Errorhandler der Bibliothek sind nämlich ebenfalls mit *catch/throw*-Kombinationen implementiert. Dieses hat zur Folge, daß das System bei einem Fehler innerhalb eines Threads, der nicht der *main*-Thread ist, in einem undefinierten Zustand ist. Da eine Entwicklung aber unter ACL geschieht, können Fehler schon hier erkannt und analysiert werden. Sieht man den Übersetzer jedoch als eigentliches LISP-System, so ist eine korrekte Implementierung der *catch/throw*-Konstrukte eine unbedingt notwendige Erweiterung. Eine mögliche Realisierung eines Exception-Handlers in einem parallelen LISP beschreiben Halstead und Loaizie in [HL85a].

3.5 Vergleich zu anderen LISP-Systemen

Wie in den vorangegangenen Abschnitten beschrieben, sollte es sich bei der parallelen Übersetzer-Bibliothek nie um ein vollständiges paralleles LISP handeln. Trotzdem sei hier ein kurzer Vergleich zu bestehenden parallelen LISP-Implementationen gezogen. Die grundlegenden Fragen bei diesem Vergleich sind dabei:

- Wie wird Parallelität erzeugt?
- Wie synchronisieren sich Threads?
- Welche Granularität der Parallelität ist sinnvoll?
- Welche sonstigen Erweiterungen (z.B. am Typsystem) stellt das System zur Verfügung?

Das in dieser Arbeit realisierte Interface stellt die einfachste Art einer LISP-Erweiterung dar. Nebenläufigkeit wird nur durch einen entsprechenden Funktionsaufruf *(new-thread Startfunktion Startparameter)* erzeugt. Zur Synchronisation stehen *mutex*-Variablen und *Conditions* zur Verfügung, jedoch mit der Einschränkung, daß diese nur auf der äußersten Programmebene in Form von globalen Variablen definiert werden können. Threads können für beliebig kleine Befehlsfolgen erzeugt werden, die Zeit für das Erzeugen und Beenden von Threads (siehe dazu auch Anhang B) lassen eine zu feinkörnige Wahl jedoch nicht sinnvoll erscheinen.

Eine etwas abstraktere Art der Erzeugung von Nebenläufigkeit bietet das **Future**-Konzept, erstmals in Multilisp [HL85b] implementiert. Die Idee dabei ist, Werteberechnungen schon wesentlich eher einzuleiten als sie benötigt werden. Damit muß nur dann auf das Ende einer Berechnung gewartet werden, wenn der Wert zum Zeitpunkt der Weiterverwendung noch nicht berechnet ist.

```
(setf a (future (+ 1 2)))    ;;Erzeugt einen Thread zur Auswertung von (+ 1 2)
(setf b (future (* 3 4)))    ;;Erzeugt einen Thread zur Auswertung von (+ 3 4)
(do-somehing)                ;;setze Programm fort ohne a und b zu benutzten
```

[8] Dieses ist ein spezielles C-Konstrukt. Die *setjump()*-Funktion erzeugt eine Markierung an der Stelle im Programm, an der sie aufgerufen wurde. Mit Hilfe der *longjump()*-Funktion läßt sich von beliebiger Stelle des weiteren Programmverlaufs aus zu dieser Markierung zurückspringen.

```
(+ a b)                    ;;Hier muss auf das Ergebnis gewartet werden, falls
                           ;;ein Thread noch nicht fertig ist
```

Das Warten auf das Ergebnis (Ende des Threads) kann für den Programmierer transparent implementiert werden. Ein *Future*-Aufruf liefert einen sog. **Placeholder** zurück. Dieser hat anfangs den Status „unaufgelöst“. Ist der zugehörige Thread mit seiner Berechnung fertig, so wird der *Placeholder* in den entsprechenden Wert der Berechnung umgewandelt. Bei der Verarbeitung von *Placeholdern* teilen sich die LISP-Funktionen in zwei Klassen auf. Die **nicht-strikten Funktionen** wie Listenoperationen benötigen den Wert des *Placeholders* nicht. Sie können also damit arbeiten, ohne auf den dazugehörigen Thread warten zu müssen. **Strikte Operationen** wie arithmetische Operationen oder Vergleiche benötigen den Wert des *Placeholders*. Sie müssen daher warten, bis die *Futures* aufgelöst sind. Nur bei strikten Operationen müssen sich die Threads daher synchronisieren. Der Programmierer sollte also wissen, welche Operationen als nicht-strikt und welche als strikt gelten. Als weitere Synchronisationsmechanismen stehen *mutual-exclusion*-Befehle zur Verfügung. Auch unter Multilisp lassen sich einzelne LISP-Befehle als eigene Threads abarbeiten. Die Kosten für die Erzeugung schränken die praktische Verwendung jedoch ebenfalls ein.

Einen ganz anderen Weg zur Erzeugung von Nebenläufigkeit geht die Connection-Machine(CM) LISP-Implementation [SH86]. Grundlage aller Parallelkonstrukte ist der Datentyp *xapping*. Ein Objekt vom Typ **xapping** ist eine ungeordnete Menge geordneter Paare. Ein Paar wird dabei in der Form ($index \rightarrow value$) notiert. Sowohl $index$ als auch $value$ können dabei beliebige LISP-Objekte sein. Nebenläufigkeit wird dann durch die Anwendung von Funktions-*xappings* auf Argument-*xappings* erzeugt. Ein Beispiel zeigt die Anwendung.

<u>Definition der xappings:</u>
worldinfo-1 = {sky→blue, apple→red, grass→green}
worldinfo-2 = {sky→cloudy, apple→sweet, grass→wet, weather→good}
combine-infos = {sky→cons, earth→list, apple→cons}

<u>Anwendung:</u>
(combine-infos worldinfo-1 worldinfo-2) ⇒ {sky→(blue.cloudy), apple→(red.sweet)}

Für den häufig auftretenden Fall, daß auf alle *xapping*-Elemente dieselbe Funktion angewendet werden soll, kann man dieses mit dem α-Operator abkürzen. $\alpha cons$ expandiert z.B. um einen *xapping*, indem für jeden möglichen Index I ein Element $I \rightarrow cons$ steht. Auf die Argument-*xappings* des Beispiels angewandt, würde der Befehl $\alpha cons$ folgendes bewirken:

(αcons worldinfo-1 worldinfo-2) ⇔ ({sky→cons, apple→cons, grass→cons, weather→cons} worldinfo-1 worldinfo-2)
⇒ {sky→(blue.cloudy), apple→(red.sweet), grass→(green.wet)}

Die Nebenläufigkeit wird hier also datengesteuert erzeugt und nicht wie bei den anderen Systemen programmgesteuert. Dieses ist der typische Ansatz bei einer SIMD-Architektur. Eine explizite Synchronisation durch den Programmierer ist hier nicht notwendig. Bei einer SIMD-Architektur wie der Connection-Machine wird die Synchronisation auf der Hardwareebene durchgeführt. Wird dieser datengesteuerte Ansatz auf einer MIMD-Architektur verwandt, so ist eine implizite Synchronisation beim Aufsammeln der einzelnen Werte für das Ergebnis-*xapping* notwendig.

Alle drei Ansätze geben dem Programmierer die Kontrolle über die Verwendung von Threads.

Die datengesteuerte Version erfordert jedoch ein Umdenken bei der Entwicklung, so daß vor allem die Parallelisierung bestehender Programme in den meisten Fällen mit der programmablauforientierten Version einfacher ist. Die Frage, ob man dem Entwickler *low-level*-Konstrukte wie in dieser Implementierung oder abstrakte Konstrukte – wie das *future*-Konzept – zur Verfügung stellt, ist eher eine philosophische Frage. Für eine abstrakte Version spricht die einfachere Handhabung. Die Transparenz wird aber dadurch erkauft, daß es z.B. durch unbedachte implizite Synchronisation zu überraschenden Performanceeinbußen kommen kann [Kes90]. Eine *low-level*-Implementation hält die Option offen, abstrakte Konstrukte auf ihr aufzubauen und gibt dem Programmierer die volle Kontrolle, aber auch die volle Verantwortung.

4

STRUKTURANALYSE GESPROCHENER SPRACHE MIT EINEM CHART-PARSER

Das Ziel der Sprachverarbeitung mit Computern ist das „Verstehen“ der eingegebenen Äußerung. Görz beschreibt dieses Verstehen in [Gör88, S. 111] als Überführung einer Wortfolge oder Menge von Worthypothesen „in einen formalen Ausdruck, der ihre Bedeutung(en) auf dem Hintergrund semantischen und pragmatischen Wissens darstellt“. Ein Schritt in dieser Überführung ist die syntaktische Analyse. Sie untersucht, ob eine zusammenhängende Folge von Worten (Satz) als grammatikalisch korrekt gilt. Im Hinblick auf eine Weiterverarbeitung ist dabei besonders die während des Analysevorgangs aufgebaute Struktur wichtig. Sie soll den Aufbau des Satzes aus seinen Bestandteilen (Konstituenten) repräsentieren. Die Grundannahme ist dabei, daß sich eine Grammatik natürlicher Sprache –oder zumindest ein hinreichend großer Ausschnitt aus ihr– in derselben Klasse von Grammatiken beschreiben läßt, in der formale Sprachen wie z.B. Programmiersprachen beschrieben werden. Obwohl es sich nach dieser Annahme bei der Strukturanalyse eines Programmtextes und eines gesprochenen Satzes also prinzipiell um das gleiche Problem handelt, sind doch die Randbedingungen sehr unterschiedlich. Daher gibt es für verschiedene Bereiche auch verschiedene Ansätze, das Problem zu lösen. Wie dieses Kapitel zeigen wird, erweist sich der von Kay [Kay73] entwickelte Chart-Parser als besonders geeignetes Mittel zur Analyse gesprochener und geschriebener Sprache.

4.1 Idee des Chart-Parsing

Viele Algorithmen für die Syntaxanalyse arbeiten nach dem Backtracking-Verfahren, wenn alternative Fortsetzungen der bisherigen Analyse möglich sind. Trifft ein Algorithmus auf einen solchen Punkt, so wird der momentante Zustand der Anaylse zwischengespeichert. Scheitert eine mögliche Fortsetzung, wird der gespeicherte Zustand wiederhergestellt und eine der nächstmöglichen Fortsetzungen getest. Die in der verworfenen Fortsetzung erkannten Teilanalysen müssen, falls sie für weitere Fortsetzungen ebenfalls interessant sind, wieder neu analysiert werden. Ein Möglichkeit, diese Wiederholung zu vermeiden, ist die Benutzung einer Tabelle[1], in der vollständige Teilanalysen abgelegt werden. Falls an einer bestimmten Stelle der Analyse eine Teilanalyse benötigt wird, wird erst in der Tabelle nachgesehen, ob diese Teilanalyse schon vorhanden ist. Die Chart stellt eine Verallgemeinerung dieser Tabelle dar. Ausgehend von einer kontextfreien Grammatik soll an einem Beispiel des einfachen Satzes „Das Kind sieht den Vogel“ das Verfahren der Chart-Analyse beschrieben werden.

[1]Man spricht von einer *wellformed substring table*.

Die kontextfreie Grammatik G sei gegeben durch das Quadrupel (NT,T,P,s):

NT={NP,VP,S,V,N,DET} Menge der Nicht-Terminale
T={das, Kind, singt, hört, den, Vogel} Menge der Terminale
P={S $\rightarrow$ NP VP,
NP $\rightarrow$ DET N,
VP $\rightarrow$ V | V NP,
N $\rightarrow$ Kind | Vogel,
DET $\rightarrow$ das | der,
V $\rightarrow$ singt | sieht} Menge der Regeln aus $N \times \{NT \cup T\}$
s=S Startsymbol der Grammatik

Die Chart besteht aus Knoten und Kanten. Für jeden Wortzwischenraum sowie Anfang und Ende des Satzes wird ein Knoten eingeführt. Diese werden zur besseren Referenzierung von 1 bis n+1 durchnumeriert (n sei die Zahl der Wörter im Satz). Für jedes Wort wird eine Kante zwischen seinem Anfangs- und Endknoten gezogen. Die initiale Chart für die Analyse des Beispielsatzes sieht damit, wie in Abbildung 4.1 dargestellt, aus.

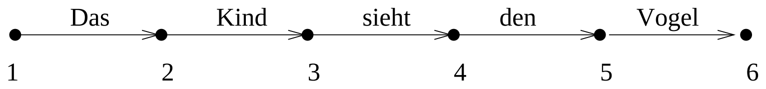


Abbildung 4.1 Initiale Chart für den Beispielsatz

Im ersten Analyseschritt werden die Terminale auf ihre entsprechenden Kategorien abgebildet, d.h. für jede Kante werden weitere Kanten gezogen, die den gleichen Anfangs- und Endknoten haben, aber mit den entsprechenden Kategoriesymbolen gekennzeichnet sind (Abbildung 4.2).

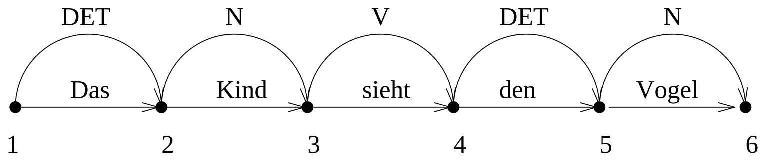


Abbildung 4.2 Chart mit Kategoriekanten

Die so entstandenen Kanten können jetzt mit Hilfe der passenden Produktionen zu längeren, d.h. mehrere Knoten überspannenden Kanten erweitert werden, bis keine Kombinationsmöglichkeit mehr vorhanden ist. Ist der Satz syntaktisch korrekt, so enthält die Chart mindestens eine alle Knoten überspannende Kante, die das Startsymbol der Grammatik als Kategorie hat. Vermerkt man zusätzlich bei jeder neuen Kante, aus welchen Kanten diese enstanden ist, so läßt sich aus der Chart die komplette syntaktische Zerlegung des Satzes in Form eines Ableitungsbaumes ermitteln. Wie die Abbildung 4.3 zeigt, wird beim Weiterspannen einer Kante nicht die Originalkante verlängert, sondern eine neue Kante eingeführt. Den Vorteil, den das Aufheben der Originalkante bietet, wird im folgenden noch deutlich.

In der bisherigen Form ist die Chart eine andere Darstellung der beschriebenen Tabelle. Ein aktiver Chart-Parser stellt aber nicht nur die vollständigen , sondern auch alle unvollständigen Teilanalysen in der Chart dar. Um eine unvollständige Teilanalyse zu beschreiben, wird eine weitere Kantenart benötigt. Die bisher verwendeten Kanten sind sog. inaktive Kanten, d.h. die Regel, die zur Bildung

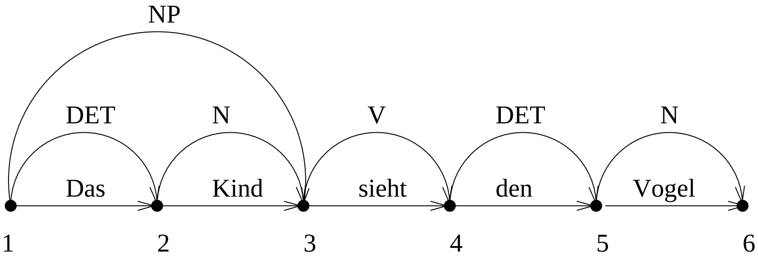


Abbildung 4.3 Chart mit einer weitergespannten Kante

einer Kante führt, ist vollständig abgearbeitet. Aktive Kanten sind dagegen Kanten, deren Produktion noch nicht vollständig abgearbeitet wurde. Als Markierung zwischen den analysierten und den fehlenden Konstituenten der Regel wird ein Punkt verwendet.[2] Z.B. bedeutet NP $\rightarrow$ DET$\bullet$N, daß die zugehörige aktive Kante die Kategorie NP besitzt und schon eine Determinante überspannt, ihr aber noch ein Nomen fehlt. Steht der Punkt vor der ersten gesuchten Konstituente, so spricht man von einer **leeren aktiven Kante**. Für jede gesuchte Kategorie wird also in den entsprechenden Knoten als erstes eine Menge von leeren aktiven Kanten eingetragen, deren Abarbeitung zu der gewünschten Kategorie führt.

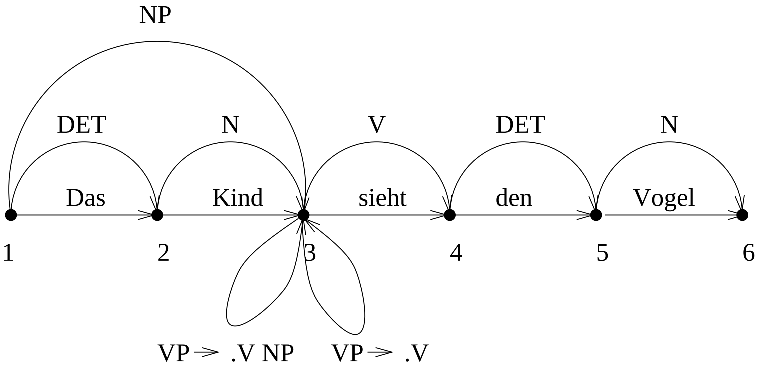


Abbildung 4.4 Chart mit aktiven Kanten

Trifft nun eine aktive auf eine inaktive Kante, die mit der nächsten gesuchten Kategorie gekennzeichnet ist, so kann man die aktive Kante um die Knoten, die die inaktive überspannt, erweitern und den Punkt eine Konstituente weiter nach rechts schreiben. Ist keine weitere Konstituente mehr vorhanden, so wird die Kante inaktiv, sonst bleibt sie aktiv. Lassen sich keine aktiven und inaktiven Kanten mehr kombinieren, so ist die Analyse abgeschlossen und alle möglichen syntaktischen Zerlegungen des Satzes stehen in der Chart.

Die Chart-Analyse bietet folgende Vorteile für die Sprachverarbeitung:

1. In der Chart können sowohl lexikalische als auch syntaktische Ambiguitäten ohne Probleme dargestellt werden. Erkannte Konstituenten, die in verschiedene Analysen einfließen, werden nur einmal repräsentiert. Ein Redundanzcheck vor dem Eintragen einer Kante garantiert, daß keine zwei gleichen Kanten in der Chart existieren.

2. Das Schema gibt im Gegensatz zu anderen Verfahren weder eine spezielle Suchreihenfolge noch eine bestimmte Analysestrategie vor. Durch eine Parametrisierung sind verschiedene Analyse-

[2]Die Regeln werden daher auch als *dotted rules* bezeichnet.

strategien realisierbar. Insbesondere kann während des Analysevorgangs die Strategie gewechselt werden. Auf Such- und Analysestrategien wird im Abschnitt 4.3 näher eingegangen.

3. Auch wenn kein vollständiger Satz erkannt worden ist, so stehen doch Teilanalysen zur Verfügung, die evtl. durch weitere Wissensquellen (Semantik, Pragmatik) zu einer korrekten Erkennung der Äußerung führen können. Dieses gilt vor allem für eine später noch näher erläuterte Bottom-Up Strategie.
4. Konkurrierende Worthypothesen, wie sie in gesprochener Sprache vorkommen, können verarbeitet werden.
5. Eine inkrementelle Analyse der Eingabe ist möglich.[3] Weil alle aktiven Kanten aufgehoben werden, kann eine nachträglich eingefügte Wortkante ohne weitere Veränderungen am Algorithmus in die Analyse einfließen.

Im nächste Abschnitt wird ein Algorithmenschema für einen abstrakten Chart-Parser nach Winograd [Win83] definiert, der als Grundlage einer Implementierung dienen kann.[4]

4.2 Ein abstrakter Chart-Parser

Formal gesehen ist eine Chart ein gerichteter Graph, beschrieben durch das Paar (V, E). V ist dabei eine endliche, nichtleere Menge von Knoten und $E \subseteq V \times V \times R$ eine endliche Menge von Kanten. R ist eine endliche Menge kontextfreier Regeln. Die Datenstruktur *Chart* läßt sich in folgender Weise darstellen:

Datentyp **Aktive Chart**

Beschreibung:	Aufzeichnung aller vollständigen und unvollständigen Konstituenten, die während der Syntaxanalyse eines Satzes entstehen
Komponenten:	Menge von Knoten
	Menge von Kanten
Subtypen:	Datentyp **Knoten**, Datentyp **Kante**

Datentyp **Knoten**

Beschreibung:	Eine Stelle in der zu parsenden Eingabesequenz
Komponenten:	Menge von eingehenden Kanten
	Menge von ausgehenden Kanten

Datentyp **Kante**

Beschreibung:	Darstellung einer vollständigen oder unvollständigen Konstituente
Komponenten:	Anfangsknoten (Knoten in der Chart)
	Endknoten (Knoten in der Chart)
	Kategorie (Nichtterminal der Grammatik)
	Fortsetzung (eine evtl. leere Folge von Symbolen aus N×T)
Zustand:	{aktiv, inaktiv}
Prädikate:	Gleichheit (gleiche Anfangs- und Endknoten, sowie gleiche Kategorie und gleiche Fortsetzung)

[3] Zur genauen Definition und Realisierung der Inkrementalität siehe Abschnitt 4.5.

[4] Weitere Beschreibungen finden sich in [Gör88] und [TR84].

Für die Anlyse wird zusätzlich eine Datenstruktur benötigt, die die zur Bearbeitung anstehenden Kanten aufnimmt:

Datentyp **zur Bearbeitung anstehende Kanten**	
Beschreibung:	Aufzeichnung aller noch zu bearbeitenden Kanten
Komponenten:	Menge von Kanten

Folgende Operationen auf der Chart lassen sich festlegen:

Funktion	Beschreibung
INIT()	Initialisierung der Chart: Erzeuge für die Eingabe von n Wörtern n+1 Knoten. Trage für jede lexikalische Kategorie jedes Wortes eine Kante in die Chart ein. Trage eine leere aktive Kante mit dem ausgezeichneten Startsymbol ein.[5] Wende PROPOSE() auf alle erzeugten Kanten an. Diese führt zu neuen Kanten, die zur Bearbeitung anstehen.
COMBINE(*aktiv, inaktiv*)	Falls der Endknoten von *aktiv* gleich dem Anfangsknoten von *inaktiv* ist, und die Kategorie von *inaktive* die nächste gesuchte Kategorie von *aktiv* ist, so erzeuge eine neue Kante auf folgende Weise: ▪ Ihr Anfangsknoten ist der Anfangsknoten der aktiven Kante ▪ Ihr Endknoten ist der Endknoten der inaktive Kante ▪ Ihre Kategorie ist die Kategorie der aktiven Kante ▪ Ihre Fortsetzung ist die Fortsetzung der aktiven Kante ohne das erste Element ▪ Ist die Fortsetzung leer, so ist die neue Kante inaktiv, sonst aktiv Die neue Kante wird in die Menge der zur Bearbeitung anstehenden Kanten aufgenommen. Die COMBINE()-Funktion wird auch als **fundamentale Regel** bezeichnet.
PROPOSE(*Kante*)	Schlägt neue Kanten vor: Aufgrund des Inhalts der aktuelle Kante *Kante* werden neue, leere aktive Kanten vorgeschlagen. Alle neuen Kanten werden in die Menge der zur Bearbeitung anstehenden Kanten aufgenommen. PROPOSE() erhält im Gegensatz zur Darstellung bei Winograd die gesamte Kante als Parameter. Dadurch ist es möglich, die Parsingstrategie zentral in dieser Funktion festzulegen. Definiert man PROPOSE() nur in Abhängigkeit der Kategorie und des Knotens, so muß sowohl die Kontrollschleife als auch diese Funktion der entsprechenden Parsingstrategie angepaßt werden.

Tabelle 4.1 Operationen auf der Chart

Funktion	Beschreibung
INSERT(*Kante*)	Falls eine gleichartige Kante existiert, beende die Funktion (Redundanzcheck). Andernfalls trage die Kante *Kante* in die Chart ein. Ist die Kante inaktiv, so wende COMBINE() auf alle Paare (*aktiv, Kante*) an. *aktiv* ist eine Kante aus der Menge der aktiven Kanten, die im Anfangsknoten von *Kante* enden. Ist die Kante aktiv, wende COMBINE() auf alle Paare (*Kante, inaktiv*) an. Hier ist *inaktiv* eine Kante aus der Menge der inaktiven Kanten, die im Endknoten von *Kante* beginnen.

Tabelle 4.1 - *Fortsetzung* - Operationen auf der Chart

Das folgenden Struktogramm beschreibt die Kontrollschleife, die den Parsingprozeß steuert:

abstrakter Chart-Parser — Hauptprogramm

INIT()	
solange Kanten zur Bearbeitung anstehen	
	Wähle eine Kante K aus der Menge der zur Bearbeitung anstehenden Kanten aus
	INSERT(K)
	PROPOSE(K)

4.3 Spezialisierungen des Parsers

Der abstrakte Chart-Parser stellt ein Algorithmenschema dar, daß durch Spezialisierungen der einzelnen Funktionen zu einem Parseralgorithmus führt. Das Schema hat zwei Freiheitsgrade. Zum einen bleibt offen, wann neue Kanten vorgeschlagen und dadurch Regeln angewendet werden. Diese Festlegung bestimmt die Parsingstrategie. Zum anderen bleibt die Art der Auswahl bei Alternativen offen. Diese Auswahl legt die Suchstrategie des Algorithmus fest.

[5] Diese Kante wird nur für den Top-Down Modus benötigt, stellt für den Bottom-Up Modus aber kein Problem dar. Zu den beiden Modi siehe Abschnitt 4.3.1.

4.3.1 Parsingstrategien

Die klassischen Parsingstrategie sind Top-Down- oder eine Bottom-Up Strategie. Bei der **Top-Down** Strategie wird ausgehend von dem Startsymbol versucht, eine syntaktische Zerlegung des Eingabesatzes zu finden. Die linke Seite einer Regel wird dabei als ein Ziel angesehen, daß zu erfüllen ist, indem seine Unterziele (Konstituenten auf der rechten Seite) erfüllt werden. Man spricht daher auch von einer zielgesteuerten Analyse [Gör88]. Der Ableitungsbaum wird also von oben nach unten aufgebaut. Erreicht wird diese Strategie, indem durch die Prozedur PROPOSE() nur dann neue leere Kanten vorgeschlagen werden, wenn der Parameter eine aktive Kante ist. In diesem Fall wird für jede Grammatikregel, deren linke Konstituente mit dem ersten Element der Fortsetzung der aktiven Kante übereinstimmt, eine neue Kante vorgeschlagen. Dieses wird in Abbildung 4.5 am Beispiel der Startkante verdeutlicht.

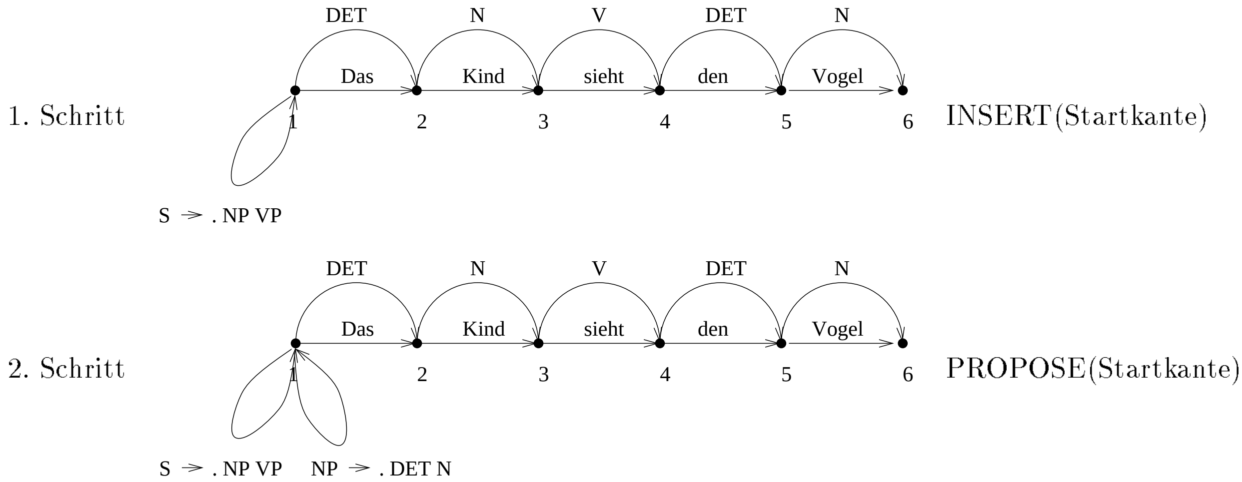


Abbildung 4.5 Einfügen der Startkante bei der *Top-down* Strategie

Im Gegensatz dazu schlägt im Fall der **Bottom-Up** Strategie PROPOSE() nur neue Kanten vor, wenn eine inaktive Kante eingetragen wird. Dabei wird die Grammatik nach Regeln durchsucht, deren erste Konstituente der rechten Seite der Kategorie der inaktiven Kante entspricht. Der Ableitungsbaum wird von unten nach oben aufgebaut, indem versucht wird, Konstituenten zu neuen, größeren Konstituenten zusammenzufassen. Dieses bezeichnet man als datengesteuerte Analyse. Abb. 4.6 zeigt die Chart nach der Initialisierung bei einer *Bottom-Up* Strategie.

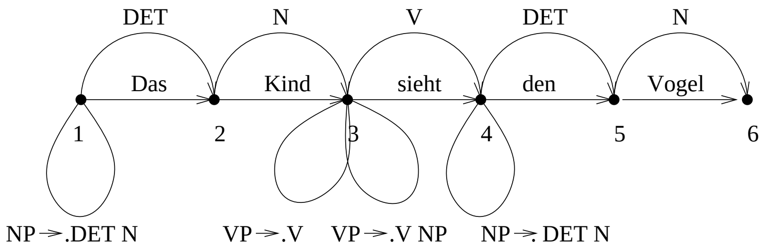


Abbildung 4.6 Chart nach dem ersten PROPOSE()-Aufruf bei der *Bottom-up* Strategie

Neben diesen beiden Strategien sind noch weitere denkbar, die aber zum größten Teil Modifikationen der beschriebenen sind [Wir92].

4.3.2 Suchstrategien

Die Kontrollschleife im Hauptprogramm bestimmt keine Reihenfolge, in der die zur Bearbeitung anstehenden Kanten zu verarbeiten sind. Ebenso ist die Auswahl von Paaren aktiver und inaktiver Kanten für die fundamentale Regel unbestimmt. Durch die Art, wie diese Auswahl getroffen wird, wird die Suchstrategie des Chart-Parsers bestimmt. Beide Auswahlmöglichkeiten stellen Aufträge an den Chart-Parser dar, die noch abzuarbeiten sind. Der Puffer, in dem diese Aufträge zwischengespeichert werden, heißt **Agenda**. Eine Suchstrategie wird durch die beiden Operationen *Auftrag einfügen* und *Auftrag entnehmen* spezifiziert. Die beiden Grenzfälle realisierbarer Suchstrategien sind die Breiten- und die Tiefensuche. Bei der **Tiefensuche** wird die Agenda als *Stack* organisiert. Der Eintrag, der zuletzt gespeichert wurde, wird zuerst entnommen. Dadurch wird eine Regel so weit wie möglich abgearbeitet, bevor eine Alternative angefangen wird. Bei der *Breitensuche* werden Einträge in Form einer Warteschlange verwaltet. Der Auftrag, der zuerst eingetragen wurde, wird auch zuerst entnommen. Es werden immer alle möglichen Regeln sozusagen synchron um einen Schritt erweitert. Aber auch beliebige andere Organisationsformen, die eine Mischform aus den beiden genannten darstellen, sind denkbar. Liegt eine Bewertung der Kanten vor, so kann eine **Beste-Zuerst Suche** realisiert werden. Bei gesprochener Sprache kann z.B. die Wahrscheinlichkeit, mit der die einzelnen Hypothesen erkannt worden sind, diese Bewertung liefern. Solange die Agenda vollständig abgearbeitet wird, sind alle Strategien gleichwertig. Das Problem der Endlosschleife, das einige Verfahren bei der Verwendung von Tiefensuche mit linksrekursiven Grammatiken haben, tritt beim Chart-Parser nicht auf. Durch den Redundanzcheck wird sichergestellt, daß nicht unendlich viele leere Kanten eingetragen werden. Das Weiterspannen einer Kante mit einer linksrekursiven Regel kann höchstens bis zum Endknoten erfolgen, so daß auch hier keine Endlosschleife entstehen kann. Für leere aktive Kanten ist der Redundanzcheck also unerläßlich, während er bei allen anderen Kanten nur zur Vermeidung doppelter Arbeit notwendig ist. Das Ergebnis wird dadurch nicht beeinflußt. Interessant wird die Agendaorganisation erst, wenn die Analyse vorzeitig abgebrochen wird oder Zwischenergebnisse möglichst schnell weitergegeben werden sollen. In diesem Fall kann das Analyseergebnis und die Zeit, in der es gefunden wird, unterschiedlich sein.

4.4 Grammatik und Unifikation

Die Grammatik aus Kapitel 4.1 läßt nicht nur korrekte deutsche Sätze zu, sondern auch falsche wie z.B. „Den Vogel singt“. Eine sinnvolle Grammatik muß daher weitere linguistische Einschränkungen enthalten, wie z.B. Kasus- und Numeruskongruenz bei der Bildung einer Nominalphrase. Aus dem Symbol NP würden dadurch acht neue Nichtterminale entstehen. Die Anzahl der Regeln würde also unter Berücksichtigung aller linguistischen Einschränkungen explosionsartig erhöhen.[6] Eine rein kontextfreie Grammatik ist daher keine geeignete Repräsentation für linguistisches Wissen. Deshalb sind zur Zeit verwendete Formalismen zum größten Teil **unifikationsbasierte Grammatikformalismen**. Die Beschreibung der syntaktischen Struktur erfolgt durch sog. **Merkmalsstrukturen**.[7] Eine Merkmalsstruktur ist eine Menge von Merkmalen. Diese wiederum bestehen aus einem **Merkmalsnamen** und einem **Merkmalswert**. Werte sind dabei entweder leer, atomar oder wieder Merkmalsstrukturen. Die Menge der Namen und der Atome ist endlich. Die Merkmalsstruktur für das Wort „Kind“ kann z.B., wie in Abb. 4.7 dargestellt, aussehen:

[6] Außerdem sind gewisse Konstruktionen in der deutschen Sprache nicht durch eine kontextfreie Grammatik beschreibbar.

[7] Daher werden diese Formalismen auch merkmalsbasiert genannt.

$$D_{Kind} = \begin{bmatrix} Cat: & N \\ Head: & \begin{bmatrix} String: & \text{“Kind”} \\ Agr: & \begin{bmatrix} Num: sing \\ Case: nom \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Abbildung 4.7 Merkmalsstruktur für das Wort “Kind” in Matrixschreibweise

Die Folge von Merkmalsnamen, die einen Wert identifizieren, bezeichnet man als **Pfad**, geschrieben A(`name1, name2, ...`). *A(Head,Agr,Num)* im obigen Beispiel ist also das Merkmal *sing*. Der **Definitionsbereich einer Merkmalsstruktur** ist die Menge aller Pfade, die genau einen Namen enthalten, d.h. dom(D_{Kind})={Cat,Head}. Verweisen zwei Pfade auf einen identischen Merkmalswert, so spricht man von einer Koreferenz. Diese wird in der Matrixschreibweise über Variablen dargestellt (Abb.4.8).

$$D_{das-Kind} = \begin{bmatrix} Cat: & NP \\ Head: & \begin{bmatrix} String: & \text{“Kind”} \\ Det: & \begin{bmatrix} String: & \text{“das”} \\ Arg: & \%1 \end{bmatrix} \\ Agr: & \%1 = \begin{bmatrix} Num: & sing \\ Case: & nom \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Abbildung 4.8 Merkmalsstruktur mit Koreferenz

Eine alternative Darstellung der Merkmalsstrukturen ist die eines **gerichteten azyklischen Graphen (Directed Acyclic Graph, DAG)**.[8] Die Merkmalswerte einer Struktur sind dann die Knoten des Graphen, die Namen Beschriftungen der Kanten.

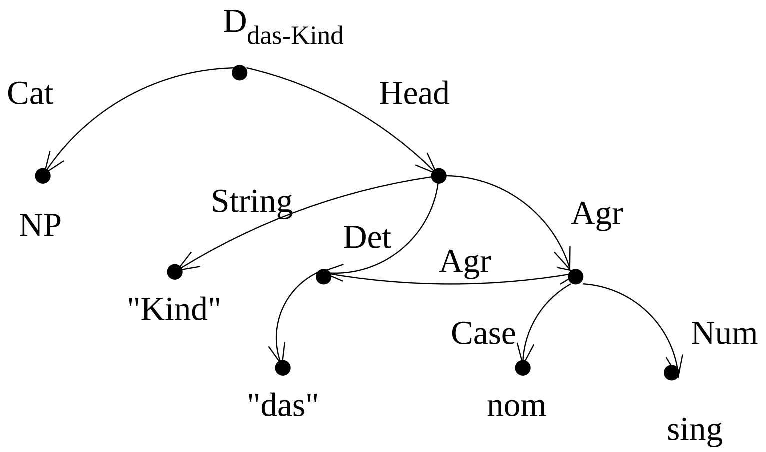


Abbildung 4.9 Graphendarstellung der Merkmalsstruktur des Wortes ”Kind“

Die wichtigste Operation, um aus gegebenen Merkmalsstrukturen neue zu erzeugen, ist die **Vereinigung** oder **Unifikation**. Dieses entspricht der Kombination der zugehörigen Graphen. Die zu

[8]Zyklische Graphen sind zwar ebenso denkbar, werden aber von den meisten Grammatikformalismen nicht verwendet. Eine sinnvolle linguistische Bedeutung scheint es darüber hinaus auch nicht zu geben (s. [Gör88, S. 154].

vereinigenden Strukturen müssen dabei kompatibel sein, d.h. unter Vereinigung und Durchschnitt wieder wohlgeformte Merkmalsstrukturen ergeben. Formal läßt sich die Unifikation mit Hilfe der **Subsumtion** definieren. Diese ist rekursiv über den Merkmalsstrukturen definiert. Sind A und B Merkmalsstrukturen, so gilt:

A subsumiert B (A$\sqsubseteq$B) $\Leftrightarrow$

- A ist der leere Graph
- A und B atomar $\Rightarrow$ A=B
- A und B komplex $\Rightarrow$

$$\forall l \in dom(A) : A(l) \sqsubseteq B(l)) \wedge (\forall p, q : A(p) = A(q) \Rightarrow B(p) = B(q))$$

Das Gleichheitszeichen stellt dabei die Identität dar. Informell kann man sagen, daß A B subsumiert, wenn A weniger Information enthält als B.

A und B sind **kompatibel**, wenn es Merkmalsstruktur D gibt, die sowohl von A als auch von B subsumiert wird:

$$\exists D : A \sqsubseteq D \wedge B \sqsubseteq D.$$

Die allgemeinste Struktur D, die diese Bedingung erfüllt, ist dann die Unifikation von A und B. Der Informationsgehalt der Ergebnisstruktur wächst also monoton mit jedem Unifikationsvorgang. Es entsteht aber keine neue Information, sondern vorhandene wird zusammengefaßt. Neue Informationen können nur das Lexikon oder die Produktionen der Grammatik liefern. Das Lexikon enthält für jedes Wort einen Eintrag, ähnlich dem in Abb.4.7. Produktionen lassen sich in Form von annotierten kontextfreien Regeln darstellen. Jeder Regel wird dabei eine Menge von Gleichungen angeheftet, die beschreiben, unter welchen Bedingungen die Regel anwendbar ist. Als Beispiel wird die bekannte Regel zur Bildung einer Nominalphrase erweitert. Die Darstellung erfolgt in einer PATR-II[9] ähnlichen Weise.

NP	$\rightarrow$ DET N[10]
NP(Cat)	= NP
N(Cat)	= N
DET(Cat)	= DET
N(Cat)	= DET(Cat)
N(Head,Agr)	= DET(Head,Agr)
NP(Head)	= N(Head)
NP(Head Det)	= DET(Head)

Das Gleichheitszeichen ist dabei nicht als ein echter Vergleich oder Gleichsetztung zu verstehen, sondern als Unifikation der jeweiligen Knoten unter den angegebenen Pfaden. Abbildung 4.8 könnte

[9] PATR-II ist eine Grammatikformalismus, der auf Shieber [Shi86] zurückgeht und Vorbild für den verwendeten Formalismus ist.

[10] Dieser Anteil der Regel wird als kontextfreies Gerüst bezeichnet

also das Ergebnis dieser Regelanwendung auf die Merkmalsstruktur D_{Kind} und einem passenden Lexikoneintrag für das Wort „das“ sein. Durch eine kleine Änderung lassen sich auch die Regeln selbst als Merkmalsstrukturen darstellen. Man kann die Symbole auch als Pfade auf der obersten Ebene einer Struktur auffassen. Die Gleichungen werden dann durch Koreferenzen beschrieben. Als Beispiel dient wieder die NP-Regel:

$$\begin{bmatrix} Np & \begin{bmatrix} Cat: & NP \\ Head: & \begin{bmatrix} String: & \%1 \\ Det: & \%2 \\ Agr: & \%3 \end{bmatrix} \end{bmatrix} \\ Det & \begin{bmatrix} Cat: & DET \\ Head: & \%2 = \begin{bmatrix} String: & [] \\ Agr: & \%3 \end{bmatrix} \end{bmatrix} \\ N & \begin{bmatrix} Cat: & N \\ Head: & \begin{bmatrix} String: & \%1 = [] \\ Agr: & \%3 = [] \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Abbildung 4.10 NP-Regel als Merkmalsstruktur

Dadurch läßt sich die gesamte Grammatik in DAGs umsetzen und jede Regelanwendung mit einem Graph-Unifikationsalgorithmus[11] realisieren.

Die Einbettung von Merkmalsstrukturen in die Chart ist sehr einfach. Die Menge der Kanten E wird um ein Element erweitert und ist nun durch $E \subseteq V \times V \times R \times Q$ beschrieben. V ist wie in Abschnitt 4.2 definiert, R stellt das kontextfreie Gerüst der Unifikationsgrammatik und Q eine potentiell unendlich große Menge von Merkmalsstrukturen dar. Um den abstrakten Chart-Parser zu erweitern, sind im wesentlichen zwei Änderungen durchzuführen. Der Gleichheitstest beim Einfügen von Kanten ist in einen Subsumptionstest umzuwandeln. Die Funktion COMBINE() testet nun nicht mehr, ob die Kategorie der inaktiven Kante die passende Fortsetzung der aktiven bildet sondern, ob ihre Merkmalsstruktur mit der nächsten, von der aktiven Kante gesuchten Teilstruktur unifizierbar ist. Der Aufwand für den Kombinationstest steigt daher beträchtlich, so daß der größte Teil der Rechenzeit des sequentiellen Parsers durch die Unifikation und durch die mit ihr verbundenen Operationen verbraucht wird. Genau diese Tatsache läßt die Parallelisierung des Parser erfolgversprechend erscheinen, da die fundamentale Regel auf ein Kantenpaar unabhängig von der Anwendung auf andere Paare ausgeführt werden kann.

4.5 Latticeparsing mit einem Chart-Parser

Im Gegensatz zur geschriebenen Sprache, bei der die eingegebenen Wörter genau bekannt sind, werden bei der gesprochenen Sprache nur Hypothesen über die im Sprachsignal enthaltenen Wörter angegeben. Ein Sprachdecoder zerlegt das Sprachsignal in gleich große Zeiteinheiten (**Frames**) und gibt zu jeder Zeiteinheit eine Menge von Wörtern aus, die innerhalb dieser Zeiteinheit enden. Jedes Wort wird dabei mit einer bestimmten Wahrscheinlichkeit erkannt. Eine **Worthypothese** besteht aus einem Anfangsframe, einem Endframe, einem Wort und einer Wahrscheinlichkeit. Zu einem

[11] Der vorliegende Parser verwendet einen Algorithmus nach Aït-Kaci, wie er in [Bac89] beschrieben ist.

Wort kann es also mehrere Hypothesen geben, die unterschiedlich viele Zeitfenster überspannen. Daher müssen die Knoten der Chart nicht mehr als Wortgrenzen, sondern als echte Anfangs- und Endfenster aufgefaßt werden. Eine Menge von Worthypothesen über einem bestimmten Zeitintervall wird als **Lattice** bezeichnet. Eine Lattice kann auch Zeitabschnitte enthalten, über denen keine Worthypothese existiert. Diese Lücken können, ebenso wie Überlappungen von Hypothesen, von einem normalen Chart-Parser nicht bearbeitet werden. Vorschläge zur Lösung dieses Problems geben Chien et al. [CCL90] und Weber [Web95]. Im weiteren Verlauf wird davon ausgegangen, daß die Lattices immer verbundene Worthypothesen (**Wortgraphen**) enthalten.[12]

Die Fähigkeit eines Sprachdecoders, nach jedem Frame eine Menge von Worthypothesen zu liefern, ist konform mit der psycholinguistischen Sichtweise des menschlichen Sprachverstehens. Ein Hörer verarbeitet eine Äußerung zeitsynchron, jedes Teilstück der Äußerung wird mit einer minimalen Verzögerung verstanden und erzeugt Erwartungen über den weiteren Verlauf der Äußerung [MWT80]. Eine Übertragung dieser Fähigkeit auf ein Sprachverarbeitungssystem bedeutet eine inkrementelle Verarbeitung des Sprachsignals. Sobald der Decoder eine Worthypothese liefert, kann ein inkrementeller Parser diese in die Chart einfügen. Der Begriff des inkrementellen Parsings wird in der Literatur unterschiedlich aufgefaßt. Wirén [Wir92, S. 2] gibt jedoch eine Definition, die eine gemeinsame Basis der Auffassungen zeigt. Er unterscheidet:

- Links-Rechts-Inkrementalität
- Volle Inkrementalität

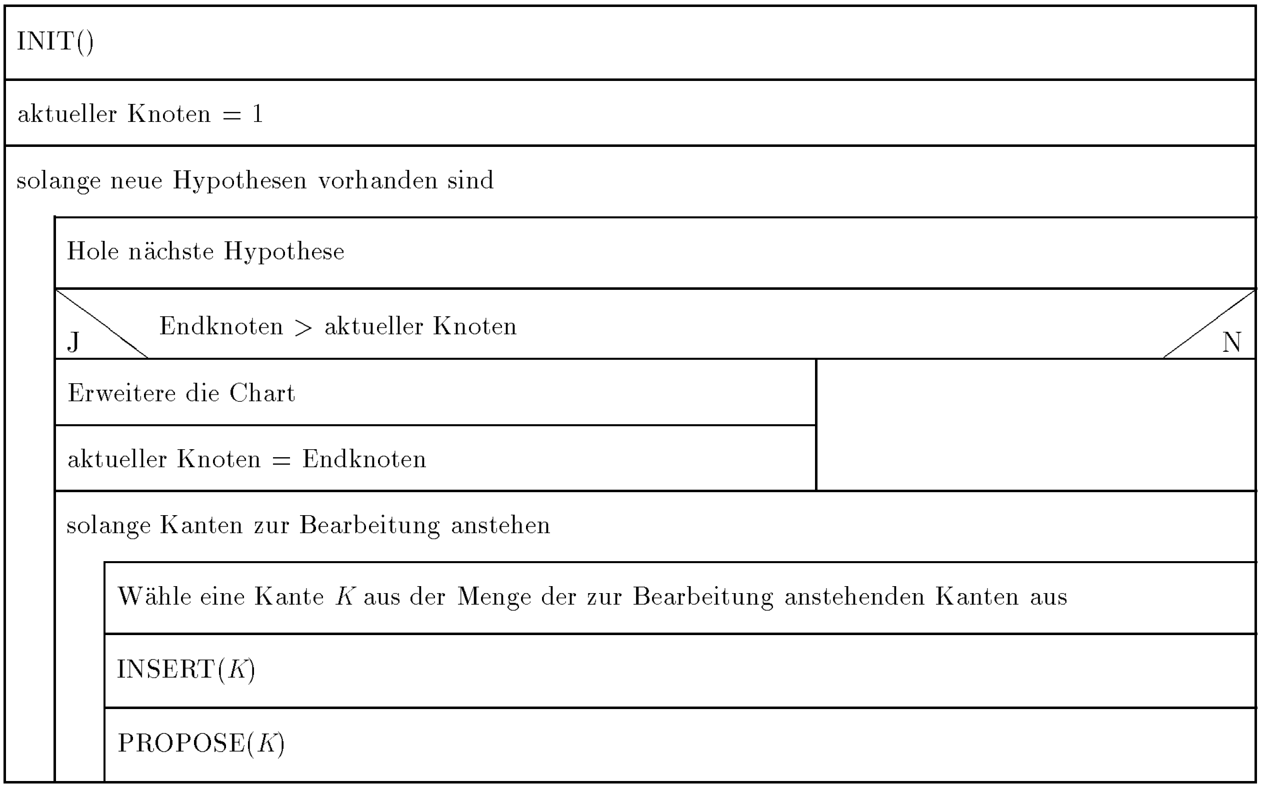
Links-Rechts-Inkrementalität ist dabei das Schritthalten des Parsers mit dem Eingabestrom. Wie im nicht-inkrementellen Fall bleibt die Monotonieeigenschaft der Chart erhalten. Volle Inkrementalität bedeutet, daß der Parser sofort auf alle Eingaben und Befehle wie Einfügen oder Löschen des Benutzers reagiert. Bei dieser Auffassung ist die Links-Rechts-Inkrementalität ein Spezialfall der vollen Inkrementalität. Wird im folgenden von Inkrementalität gesprochen, so ist immer die Links-Rechts-Inkrementalität gemeint. Bei der inkrementellen Sprachverarbeitung laufen alle Module wie Decoder, Syntax, Semantik etc. parallel. Erwartungen einzelner Module über den weiteren Verlauf der Analyse können an andere Module weitergereicht werden und so zu einer Einschränkung des Suchraums führen. Der komplette Vorgang des „Verstehens“ könnte damit sofort nach dem Ende einer Äußerung abgeschlossen sein.[13]

Die Chart wird beim inkrementellen Parsing nicht mehr vollständig, wie in Kapitel 4.2 beschrieben, initialisiert, sondern besteht am Anfang nur aus dem ersten Knoten. Sobald eine Worthypothese mit dem Endzeitpunkt entgegengenommen wird, der größer als der aktuelle letzte Knoten ist, wird die Chart um die fehlenden Knoten erweitert. Die Kontrollschleife des Parsers muß nur geringfügig verändert werden:

[12] Der in den Experimenten verwendete Sprachdecoder stellt diesen Sachverhalt sicher.

[13] Beim momentanen Stand der Entwicklung ist der Zeitbedarf der einzelnen Module noch zu hoch, als daß von „sofort“ gesprochen werden kann.

abstrakter Chart-Parser — inkrementelles Hauptprogramm



Die Ergebnischart entspricht hierbei der des Standardchartparsings bei gleicher Eingabe. Der Parser ist aber wesentlich langsamer als der Decoder.[14] Um dennoch mit dem Decoder ungefähr schritthalten zu können, darf der Parser nicht alle möglichen Pfade im Wortgraphen verfolgen. Deshalb erhält jeder Pfad eine Bewertung und alle Pfade, die eine bestimmte Schwelle (Beamschwelle) unterschreiten, werden verworfen. Die Schwelle kann dabei von außen fest vorgegeben oder dynamisch von der jeweiligen aktuellen Situation abhängig sein. Diese Verfahren nennt man **Pruning**. Pruning läßt sich sehr einfach über die Agenda realisieren. Jeder Agendaeintrag erhält zusätzlich eine Bewertung, und zwar genau die Bewertung, die die Kante bekommen würde, die aus dem Eintrag entstehen würde. Liegt diese Bewertung unter der Beamschwelle, so wird der Eintrag nicht weiter betrachtet. Die Bewertung eines Pfades kann sich aus verschiedenen Wissensquellen zusammensetzen. Die einfachste und offensichtlichste ist die Wahrscheinlichkeit, die der Decoder für den Pfad liefert. Weitere Quellen sind z.B. Wahrscheinlichkeiten für syntaktische Regeln, N-Gramm Statistiken etc.[15] Bestimmt sich der Beamwert dynamisch aus dem Wahrscheinlichkeitswert der eingehenden Worthypothesen, so würde ein Pruning jedoch nur lokal auf die Kantenkombinationen wirken, die aus der eingegangenen Hypothese entstanden sind. Daher teilt man die Hauptprozedur in drei Phasen auf. In der ersten Phase werden alle Hypothesen, die am aktuellen Knoten enden, entgegengenommen. Erst dann wird der Beamwert in Abhängigkeit von dem am besten bewerteten Agendaeintrag bestimmt. In der dritten Phase kann nun der eigentliche Parsingvorgang ausgeführt werden. Die notwendigen Modifikationen zeigt das folgende Struktogramm:

[14] Gute Decoder liegen schon im Bereich der Quasi-Echtzeit (zwei- bis dreifache Echtzeit).

[15] Der Parser benutzt außer den Decoder-Wahrscheinlichkeiten noch ein Bigramm.

abstrakter Chart-Parser — modifiziertes inkrementelles Hauptprogramm

- INIT()
- aktueller Knoten = 1
- Solange neue Hypothesen vorhanden
  - Hole alle Hypothesen des nächsten Zyklus
  - Erweitere die Chart bis zum Endknoten der Hypothesen
  - Bestimme neue Beamschwelle
  - solange Kanten zur Bearbeitung anstehen, die über der Beamschwelle liegen
    - Wähle eine Kante K aus der Menge der zur Bearbeitung anstehenden Kanten aus
    - INSERT(K)
    - PROPOSE(K)

Für eine genaue Beschreibung der Berechnung der Beamschwelle und Bewertungen sei auf [Web95] verwiesen. Diese Aufteilung der Hauptprozedur in drei Phasen hat große Auswirkungen auf die im nächsten Kapitel diskutierte Parallelisierung des Parsers.

5

PARALLELES CHART-PARSING

Syntaxanalyse in der Verarbeitung gesprochener Sprache wird immer mit dem Ziel einer Echtzeitverarbeitung betrieben. Die aktuellen Methoden und die modernen Rechnerarchitekturen erreichen dieses Ziel unter realen Bedingungen jedoch nicht. Daher liegt der Gedanke nahe, durch Parallelisierung die Analyse zu beschleunigen. Vorschläge zum parallelen Parsing mit nicht-chartbasierten Methoden finden sich in [Nij89] und [Haa87]. In diesem Kapitel wird die Parallelisierung eines Chart-Parsers auf einem *Shared-Memory* System unter besonderer Berücksichtigung der Links-Rechts-Inkrementalität untersucht.

5.1 Ansätze zur Parallelisierung

Die Chart ist die zentrale Datenstruktur des Parsers. Doch schon die Tatsache, daß die Organisation der Agenda keinen Einfluß auf das Ergebnis hat (siehe Abschnitt 4.3.2), zeigt, daß ein hoher Grad an logischer Parallelität möglich ist. Nach Amtrup [Amt92] lassen sich die möglichen Parallelisierungsansätze in vier Kategorien einteilen:

- Agendabasierte Parallelisierung

  Die Agenda wurde eingeführt, um auftretende Alternativen im Analyseprozeß in eine bestimmte Verarbeitungsreihenfolge zu bringen und so einen sequentiellen Ablauf zu ermöglichen. Für jede Alternative könnte aber auch sofort bei ihrer Entstehung ein neuer Thread zu ihrer Bearbeitung gestartet werden. Ein paralleler Parser benötigt somit eigentlich keine Agenda. Dieses entspricht der Auffassung des parallelen Parsings als ein deterministischer Prozeß mit mehreren sequentiellen Parsern (siehe [Nij89]). Da die Anzahl der Aufträge die Anzahl der Prozessoren normalerweise um ein Vielfaches übersteigt, ist der Einsatz einer Agenda als Auftragspuffer aber zu empfehlen. Der sequentielle Parser ist aus dieser Sichtweise nur ein Spezialfall des parallelen Parsers mit einem Prozessor. Diese Art der Parallelisierung ist daher die naheliegenste. Thompson [Tho84] und Grisham [GC88] erzielten bei dieser Vorgehensweise auf einem *Shared-Memory* System gute Ergebnisse. Thompson spricht gar von einem linearen *Speed-Up*.

- Knotenbasierte Parallelisierung

  Knoten können auch als Threads aufgefaßt werden. Jeder Knoten verwaltet dabei die ihm zugeordneten Kanten. Dieses können z.B. die eingehenden inaktiven und die ausgehenden aktiven Kanten sein.[1] Jeder Knoten kann nun Anforderungen an andere Knoten stellen oder Analyseer-

[1] Thompson spricht bei dem Knoten, der eine Kante verwaltet, vom *Hot-Vertex* der Kante.

gebnisse wie inaktive Kanten an interessierte Knoten mitteilen. Die Realisierung kann intuitiv durch ein *Message-Passing* System erfolgen. Kanten müssen hauptspeicherunabhängig dargestellt werden. Kritisch ist die hohe Anzahl von Nachrichten, die versendet werden müssen. Im Fall gesprochener Sprache kann auch die Knotenanzahl ein Problem darstellen. Daher werden häufig Mengen von Knoten gebildet, die zu einem Thread zusammengefaßt werden. Dabei tritt dann das *load-balancing* Problem auf. Die Knoten müssen so zusammengefaßt werden, daß möglichst viele Kanten lokal verarbeitet werden können. Thompson erzielt bei dieser Aufteilung auf einem *Workstationcluster* keinen *Speed-Up*. Amtrup berichtet in [Amt90] jedoch von guten Ergebnissen, führt diese aber zum Teil auf ein langsames Basissystem zurück. Eine weitere Implementierung auf einem Transputernetz zeigte jedoch immer noch einen *Speed-Up* von durchschnittlich 1,41 mit sieben Transputern.

- Kantenbasierte Parallelisierung

  Jede Kante stellt einen eigenen Thread dar. Aktive Kanten sind in diesem Fall echt aktive Threads, die ständig versuchen, ihre fehlenden Konstituenten zu ergänzen und dadurch neue Threads erzeugen. Inaktive Kanten haben den Charakter von Ereignissen, die bewirken, daß aktive Kanten ihren Suchvorgang fortsetzen können. Die hohe Anzahl von Kanten bzw. Threads sprengt den Rahmen bisheriger Mehrprozessorrechner, so daß an eine echte Parallelität bei diesem Ansatz nicht zu denken ist.[2] Versuche von Ruland[3] zeigten unter SOLARIS außerdem, daß die Threaderzeugungszeiten eine wesentliche Rolle bei dieser Parallelisierungsart spielen. Schon die Zeit für das Erzeugen der Threads, die für das Parsen einer natürlichsprachlichen Äußerung notwendig wären, überstieg die Gesamtlaufzeit des normalen sequentiellen Parsers.

  Faßt man Kanten nach bestimmten Kriterien zusammen, so kann man diesen Ansatz in einen knoten- oder regelbasierten Ansatz überführen.

- Grammatikbasierte Parallelisierung

  Grundlage der grammatikbasierten Parallelisierung ist das kontextfreie Gerüst der verwendeten Grammatik. Die ursprüngliche Idee stammt von Yonezawa und Ohsawa [YO89]. Sie fassen jede Konstituente einer Regel als einen eigenen Thread auf. Diese Threads schicken sich untereinander Teilanalysen zu. Aufgrund der verschiedenen Positionen an der Konstituenten in einer Regel stehen können, lassen sich die Threads in drei Typen aufteilen:

  - Typ 1: Thread, der die linke Seite einer Regel repräsentiert.
  - Typ 2: Thread, der die erste Konstituente der rechten Seite einer Regel repräsentiert.
  - Typ 3: Jede andere Konstituente.

  Jede Regel zerfällt also in einen Typ-1-Thread, einen Typ-2-Thread und eine beliebige endliche Anzahl von Typ-3-Threads. Jeder Typ-2- und Typ-3-Thread ist dabei auf eine bestimmte Konstituente abonniert. Wird nun beim Parsen eine Teilanalyse erzeugt, so erhalten diese alle Threads, die die entsprechende Konstituente der Teilanalyse abonniert haben. Das Beispiel der Regel VP$\rightarrow$V NP in Abb. 5.1 macht diese Aufteilung deutlich.

  Thread V reicht eine empfangene Teilanalyse t1 nur an Thread NP weiter. Dieser erwartet zusätzlich eine Teilanalyse t2, die eine NP darstellt und prüft, ob beide aneinandergrenzen. Im Erfolgsfall sendet er beide Teilanalysen dem VP-Thread. Dieser konstruiert dann eine beide Wortketten überspannende Analyse und sendet diese an alle Threads, die auf eine VP abonniert sind. Die Reihenfolge, in der Thread NP t1 und t2 empfängt, ist irrelevant. Er speichert

[2] Dieses wird auch auf absehbare Zeit so bleiben. Erhielte jede Kante einen Prozessor, wären mehrere tausend Prozessoren notwendig. Die für diesen Ansatz notwendigen MIMD-System sind davon aber noch weit entfernt.

[3] pers. Gespräch

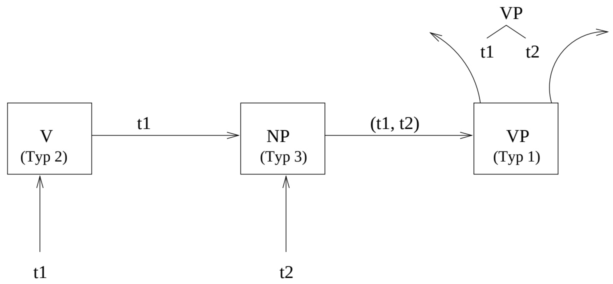


Abbildung 5.1 Darstellung einer Regel als kommunizierende Threads

die zuerst eingegangene Teilanalyse, solange bis eine passende zweite Teilanalyse eintrifft. Dies bedeutet, daß ein Thread mehrere Teilanalysen über verschiedenen Zeitabschnitten gleichzeitig gespeichert haben kann. Eine Regel, die an mehreren Stellen der Eingabe anwendbar ist, wird also nur durch einen Thread pro Konstituente dargestellt. Die Anzahl der Threads in diesem Ansatz ist für realistische Grammatiken sehr hoch. Yonezawa und Ohsawa sprechen von 1100 Threads bei einer Grammatik von ungefähr 250 Regeln. Faßt man alle Konstituenten einer Regel in einem Thread zusammen, so reduziert sich die Anzahl der Threads erheblich. Für Parallelrechner mit sehr wenigen Prozessoren ist diese Aufteilung aber noch immer zu feinkörnig. Daher werden mehrere Regeln jeweils einem Thread zugeordnet. Dabei werden die Regeln so zusammengefaßt, daß die meisten Regelanwendungen innerhalb eines Threads stattfinden. Dieses entspricht der Aufteilung einer Grammatik in mehrere Untergrammatiken, für die Weng und Stolcke in [WS93] ein formales Konzept entwickelt haben. Sie schlagen vor, für jede Untergrammatik einen eigenen Parser zu starten. Dabei kann der Algorithmus der einzelnen Parser sogar unterschiedlich sein. Amtrup hat diesen Ansatz in seiner Arbeit [Amt92] mit einem Chart-Parser auf einem Transputernetz implementiert und konnte dabei gute *Speed-Ups* erzielen. Eine Parallelisierung einer DCG (Definite Clause Grammar)[4] in Prolog zeigt Matsumoto in [Mat88].

Grundsätzlich können alle Ansätze auch als inkrementelle Parallelparser implementiert werden. Alle genannten Implementationen haben jedoch die gesamte Eingabe zum Zeitpunkt der Initialisierung vorliegen. Außerdem sind sie zwar die für Verarbeitung gesprochener Sprache vorbereitet, die Tests fanden aber immer mit geschriebenen Sätzen statt. In den folgenden Abschnitten wird daher zuerst die Tauglichkeit der Ansätze für eine inkrementelle Analyse gesprochener Sprache diskutiert, bevor auf die Realisierung einer agendabasierten Parallelisierung eingegangen wird.

5.2 Einschränkungen durch inkrementelles Parsen

Generell ist zu sagen, daß die mögliche Parallelität eines *Bottom-Up* Parsers durch einen inkrementellen Ansatz stärker eingeschränkt wird, als die eines *Top-Down* Parsers. Ein *Top-down* Parser arbeitet inhärent von links nach rechts und startet immer am ersten Chartknoten. Ein *Bottom-Up* Parser hat dagegen im nicht-inkrementellen Fall mehrere Knoten, an denen die Analyse parallel starten kann. Der nicht-inkrementelle Parser verschenkt durch eine *Top-Down* Strategie damit einen Teil seiner logischen Parallelität.

[4] siehe [PW80]

Durch den schrittweisen Aufbau der Chart ergibt sich ein besonderes Problem bei der knotenbasierten Parallelisierung. Setzt man eine Zusammenfassung der Knoten voraus, so verteilen sich die Aufträge nicht gleichmäßig auf die einzelnen Threads. Dieser Zusammenhang soll an einem Beispiel demonstriert werden.

Es wird angenommen, daß vier Threads für die Analyse existieren. Jeder Thread besitzt eine Menge von benachbarten Knoten zur Verwaltung, also z.B.

Thread 1: Knoten 1-70

Thread 2: Knoten 71-140

Thread 3: Knoten 141-210

Thread 4: alle restlichen Knoten

Knoten 1 würde also bis zum Eintreffen der ersten Hypothese, die den Knoten 70 überschreitet, alleine arbeiten. Die anderen Threads werden erst sukzessiv ihre Arbeit aufnehmen. Geht man davon aus, daß die Threads hauptsächlich lokal arbeiten – dieses ist schließlich der Sinn einer solchen Aufteilung –, so wird der Effekt der Serialisierung noch größer sein. Thread 1 hätte dann den lokalen Teil seiner Analyse schon erledigt, wenn Thread 2 seine Analyse startet. Des weiteren hat man beim inkrementellen Parsen das Problem, daß das Ende einer Äußerung nicht von Anfang an bekannt ist. Man kann daher keine gleichmäßige Verteilung der Knoten vornehmen. Denn ist die Äußerung kürzer als 70 Zeitabschnitte, so führt Thread 1 die gesamte Analyse alleine durch. Ist dagegen die Äußerung sehr viel länger als 210 Zeitabschnitte, so wird der größte Teil der Analyse von Thread 4 erledigt. Die Lastverteilung beim inkrementellen Parsen ist also sehr unausgewogen, so daß aus dieser Art der Parallelisierung keine Geschwindigkeitsvorteile zu erwarten sind. Alternativ wäre es möglich, die Knoten sukzessiv auf die Threads zu verteilen. Dieses Vorgehen würde jedoch zu einem erheblich höheren Kommunikationsaufwand führen und wäre daher nicht zielkonform. Die Knoten werden gerade deshalb zusammengefaßt, um den Kommunikationsaufwand gering zu halten.

Für die kantenbasierte Parallelisierung ergibt sich das gleiche Problem, wenn Kanten nach zeitlichen Kriterien zusammengefaßt werden. Die Lastverteilung ist durch die Inkrementalität dann sehr ungleichmäßig.

Die beiden anderen Ansätze lassen sich ohne Einschränkungen in einem inkrementellen parallelen Parser einsetzen. Die agendabasierte Parallelisierung ist auf einem *Shared-Memory* System vorzuziehen, die regelbasierte Parallelisierung kommt der Implementierung auf einem *Message-Passing* System entgegen. Weil ein *Shared-Memory* System zur Verfügung stand, wurde der agendabasierte Ansatz für die Parallelisierung des vorhandenen Chart-Parsers gewählt.

5.3 Agenda als zentrale Auftragsverwaltung

Als Grundlage der Arbeit stand der von Weber implementierte und in [Web95] beschriebene Parser in seiner *Top Down* Version zur Verfügung. Dieser arbeitet im wesentlichen nach den in Abschnitt 4.5 beschriebenen Phasen: „Hypothesen empfangen“, „Beamschwelle bestimmen“ sowie „Parsen“. Das Bestimmen der Beamschwelle erfordert eine zentrale Instanz. Daher muß der parallele Parser einen ausgezeichneten Thread (**Master**) besitzen, der erkennt, daß alle Hypothesen eines Zyklus empfangen wurden, und der dann die Beamschwelle bestimmt. Danach kann der eigentliche, parallellaufende Parsingvorgang gestartet werden. Hypothesen könnten ebenfalls parallel entgegengenommen werden. Die durchgeführten Versuche dazu haben jedoch ergeben, daß der Synchronisationsaufwand im Verhältnis zum Rechenaufwand zu hoch ist. Die notwendigen Operationen wie Suchen im Lexikon, Erzeugen der entsprechenden Kanten und evtl. Vorschlagen neuer Kanten sind durch Hashtabellen, Vorkompilation, etc. so effizient, daß in dieser Phase keine zeitaufwendigen Operationen wie Unifikation durchgeführt werden.

Der sequentielle Parser verwaltet auf der Agenda nur Paare aktiver und inaktiver Kanten. Erzeugte Kanten werden sofort eingetragen. Die Flexibilität des abstrakten Parsers bleibt aber erhalten, da man jedes Agendapaar als Repräsentant der aus ihm resultierenden Kante sehen kann. Die Parallelität des Parsers beschränkt sich also auf die nebenläufige Ausführung der fundamentalen Regel. Die Paare, auf die die fundamentale Regel angewendet wird, werden auf der sortierten Agenda abgelegt. Jedes Paar bedeutet einen Berechnungsauftrag und kann einem Thread (**Slave**) zugeordnet werden, der die COMBINE()-Funktion auf das Paar anwendet. Dabei sind zwei Zuordnungsstrategien denkbar:

1. Für jeden Auftrag wird ein eigener Thread gestartet. Dabei wird aber darauf geachtet, daß nur soviele Threads gleichzeitig gestartet werden, wie freie Prozessoren vorhanden sind. Sobald also ein Thread terminiert, und noch Aufträge mit einer Bewertung über der Beamschwelle existieren, kann ein neuer Thread gestartet werden.
2. Am Anfang wird pro Prozessor ein Thread gestartet, der darauf wartet, Aufträge von der Agenda entnehmen zu können. Sobald er einen Auftrag abgearbeitet habt, holt er sich den nächstmöglichen. In diesem Fall kann man die Threads als Dämonen[5] auffassen, die aktiv werden, sobald ein Auftrag anliegt.

Aufgrund der hohen Threaderzeugungszeiten für das LISP-System scheidet die erste Möglichkeit der Realisierung aus.

Unabhängig davon, wie Aufträge Threads zugeordnet werden, ist die grundlegende Frage der agendabasierten Parallelisierung, welche Synchronisationen für Zugriffe auf die Agenda und die Chart notwendig sind und wie groß ihr Anteil an der Gesamtlaufzeit ist.

Jeder Zugriff auf die Agenda verändert ihren Inhalt. Einfüge- und Entnahmeoperationen müssen daher exklusiv von jedem Thread ausgeführt werden. Bei der Chart kann man zwischen einem lesenden Zugriff, z.B. wenn alle ausgehenden Kanten eines Knotens gesucht werden, und einem schreibenden Zugriff unterscheiden. Der schreibende Zugriff entspricht dem Eintragen einer Kante in die Chart. Kanten sind statische Objekte, d.h. einmal erzeugt, wird ihr Inhalt nie verändert. Knoten hingegen sind dynamisch. Die Menge der ein- und ausgehenden Kanten kann ständig wachsen. Eine Leser/Schreiber-Sperre pro Knoten gewährleistet also die größte Parallelität. Folgende Argumente sprechen jedoch gegen diese Art der Synchronisation:

1. Der hohe Aufwand der Synchronisation.

   Die Threads müssen für jedes Agendapaar mehrere Sperren akquirieren und wieder freigegeben. Dieses kostet viel Zeit, da Leser/Schreiber-Sperren im Vergleich zu Mutex-Sperren teuer sind (siehe [SUN94, S. 77]).

[5] In Analogie zu UNIX-Dämonen.

2. Aufgrund der inkrementellen Analyse enden alle neu erzeugten Kanten im letzten Knoten der Chart. Schreiberanforderung treten hier also besonders häufig auf.

3. Die rekursive Implementation des Parsers erschwert die Verwendung von Sperren, da diese nicht erkennen, ob eine Sperranforderung von einem Thread kommt, der diese Sperre schon akquiriert hat. Dieses ist ein rein technischer Aspekt, da mit Hilfe der vorhandenen Sperren rekursive Sperren implementierbar sind. Auch eine nicht-rekursive Implementation des Parsers ist realisierbar. Durch die beiden zuvor genannten Argumente scheint diese aber nicht besonders erfolgversprechend.

Schwächer abgestufte Sperren würden den Grad der Parallelität zugunsten geringerer Synchronisation zurückschrauben. Das zweite Argument behält aber seine Gültigkeit. Um das Problem des hochfrequentierten letzten Chartknotens zu lösen, geht die Implementation einen anderen Weg. Man teilt die Chart in einen globalen und einen lokalen Teil ein. Bezeichne t den aktuellen letzten Knoten. Dann besteht der globale Teil der Chart aus der gesamten bis zum Knoten $t-1$ erstellten Chart. Jeder Slave hat zusätzlich einen lokalen Puffer, in den er alle neu erzeugten Kanten einträgt. Diese Puffer stellen den lokalen Teil der Chart dar, den jeder Slave aktuell zur Chart bis zum Knoten t beiträgt. Erst am Ende eines Parsingzykluses sammelt der Masterthread alle in den Puffern stehenden Kanten auf und trägt sie in die globale Chart ein, die nun bis zum Knoten t geht. Dadurch, daß alle Schreiboperationen in einen lokalen Puffer gehen, braucht die globale Chart nicht gesperrt zu werden. Nur der Masterthread kann in die globale Chart schreiben, wenn er sicher sein kann, daß kein Slave arbeitet. Der *Master* kann sich einen Prozessor mit einem *Slave* teilen, da er nur aktiv wird, wenn alle *Slaves* in einer Warteposition sind.

paralleler Parser — Master

	Hypothesen entgegennehmen
	Beamschwelle bestimmen
	Slaves signalisieren, daß Aufträge anliegen
	Warten, bis Slaves fertig
	Kanten aus den lokalen Puffern in die Chart eintragen
Keine Aufträge mehr über der Beamschwelle und keine neuen Hypothesen	

paralleler Parser — Slave

- Endlosschleife
  - Warten auf Signal des Masters
  - Aufträge über der Beamschwelle
    - Auftrag von der Agenda entnehmen
    - Auftrag ausführen
  - Master Zyklusende signalisieren

Die Idee, die hinter diesem Ansatz steht, beruht auf der Tatsache, daß das Eintragen einer Kante in die Chart eine sehr schnelle Operation ist.[6] Der sequentielle Anteil der Gesamtlaufzeit wird also zugunsten einer einfacheren Synchronisation erhöht. Eine Voraussetzung muß jedoch gegeben sein, damit dieses Verfahren korrekt arbeitet: Die Grammatik darf keine leeren inaktiven Kanten erzeugen. Diese werden häufig für einen Verbspurmechanismus eingesetzt. Werden inaktive leere Kanten erzeugt, so kann es vorkommen, daß Paare nicht auf die Agenda gelegt werden. Erzeugt ein Slave eine inaktive leere Kante i im Knoten t, so legt er sie in seinem lokalen Puffer ab. Erzeugt ein anderer Slave gleichzeitig die leere aktive Kante a, die mit i kombiniert werden könnte, so findet er bei der Suche nach neuen Paaren (aktiv, inaktiv) diese nicht, da sie unzugänglich im Puffer des anderen Slave gespeichert ist. Die Kante a würde damit trotz möglicher Verlängerung nicht weitergespannt. Ein falsches Ergebnis könnte die Folge sein. Inaktive leere Kanten müssen daher immer allen Slaves zugänglich gemacht werden. Die hier verwendete Grammatik erzeugt jedoch keine leeren inaktiven Kanten, so daß der Parser keinen Mechanismus für das Verteilen inaktiver leerer Kanten vorsieht. In allen anderen Fällen arbeitet der Parser korrekt, was sich mit einem informellen Beweis zeigen läßt.

Satz: Der parallele Parser erzeugt alle Paare (a, i) für der Knoten t, die der sequentiellen Parser für diesen erzeugt.
Voraussetzung: Es gibt keine leeren inaktiven Kanten. Die Agenda ist bis zum Zeitpunkt $t - 1$ vollständig abgearbeitet.
Beweis:

1. Der parallele Parser erzeugt alle direkt aus den empfangenen Worthypothesen entstehenden Paare, weil hier keine Änderung zum sequentiellen Parser vorliegt.

2. Führt ein Paar (a, i) zu einer neuen Kante k, so erzeugt der parallele Parser alle daraus resultierenden Paare, die der sequentielle Parser erzeugt.
   (a) a ist eine leere aktive Kante $\Rightarrow$ i ist eine leere inaktive Kante, denn der Endknoten von a ist der Anfangsknoten von i, aber auch gleichzeitig der letzte Knoten der Chart $\Rightarrow$ Paare dieser Art können nicht auf der Agenda liegen.

[6] Im günstigsten Fall sind es wenige Zeigeroperationen.

(b) a ist eine aktive Kante, deren Anfangsknoten kleiner als t ist. Dann ist auch der Anfangsknoten von k kleiner als t.

i. Ist k nun inaktiv, so besagt die fundamentale Regel, daß alle aktiven Kanten, deren Endknoten identisch mit dem Anfangsknoten von k ist, zusammen mit k als neue Paare auf die Agenda zu legen sind. Da die Chart aber bis zum Knoten $t-1$ vollständig abgearbeitet ist, kann kein Slave eine aktive Kante im lokalen Puffer haben, die im Anfangsknoten von k endet. Alle notwendigen aktiven Kanten stehen also in der globalen Chart.

ii. Ist k aktiv, so müssen alle inaktiven Kanten, die mit k neue Paare bilden könnten, in t anfangen, aber auch enden $\Rightarrow$ die inaktiven Kanten müssen leere inaktive Kanten sein $\Rightarrow$ Paare dieser Art kann es nicht geben.

Der parallele Parser kann jedoch eine größere Anzahl Kanten als der sequentielle erzeugen. Der Redundanzcheck beim Eintragen einer neuen Kante wird nicht mehr auf der Chart ausgeführt, sondern nur im lokalen Puffer jedes Slaves. Erzeugen nun mehrere Slaves lokal die gleiche leere aktive Kante, so ist sie nach dem Übertragen in die globale Chart mehrfach vorhanden. Dieses führt zu keiner falschen Lösung, erhöht aber den Parsingaufwand erheblich. Die Lösung dieses Problems bringt ein zweiter Redundanzcheck bei der Übertragung der lokalen Puffer in die Chart. Die Effizienz des Redundanzchecks ist also von großer Bedeutung für die Effizienz des parallelen Parsers. Der sequentielle Parser realisiert nur den notwendigen Check für leere aktive Kanten. Grammatikregeln sind im Parser numeriert und lassen sich über diese Nummern eindeutig identifizieren. Da leere Kanten nie eine im Verlauf der Analyse entstandene Merkmalsstruktur besitzen, läßt sich ihre Gleichheit durch einfachen Vergleich der zugehörigen Regelnummern bestimmen. Leere aktive Kanten können nur im aktuell letzten Knoten eingetragen werden. Der Redundanzcheck bezieht sich damit nur auf diese Knoten und läßt sich über ein globales Feld mit je einem Element für jede mögliche Grammatikregel realisieren. Wird eine leere aktive Kante eingetragen, so wird dieses im Feldelement mit der Nummer der assozierten Regel vermerkt. Soll die Kante ein weiteres Mal eingetragen werden, zeigt der Vermerk, daß sie schon existiert. Der Redundanzcheck beschränkt sich damit auf einen einzigen Feldzugriff. Beim parallelen Parser kann dieses Feld global oder threadlokal sein. Ein globales Feld erfordert einen synchronisierten Zugriff, verhindert aber das mehrfache Erzeugen leerer aktiver Kanten weitgehend. Da die Programmfolge *Redundanztest*, *Kante eintragen* keine unteilbare Abfolge ist, kann es passieren, daß mehrere Slaves quasi gleichzeitig den Redundanzcheck für dieselbe leere aktive Kante durchführen, alle zu einem negativen Ergebnis kommen und daher jeder die Kante in seinen Puffer aufnimmt. Der zweite Redundanzcheck des Masters kann daher nicht entfallen. Ist das Feld threadlokal, so ist die Anzahl mehrfach erzeugter Kanten wesentlich höher. Die Auswirkungen auf die Laufzeit und Effizienz solcher unterschiedlichen Strategien lassen sich nicht im voraus bestimmen. Im nächsten Abschnitt wird daher das Verhalten verschiedener Konfigurationen untersucht.

5.4 Ergebnisse

Die folgenden Tests wurden auf einem SPARCserver 1000 mit sechs Prozessoren unter SOLARIS 2.4 ausgeführt. Als Eingabe dienten jeweils zehn Lattices, die vorher von einem Sprachdecoder erzeugt worden sind. Die Laufzeit des Decoders hat also keinen Einfluß auf die Messungen. Die Grammatik enthält 34 Regeln, das Lexikon 363 Einträge.

Normalerweise sollte für Zeitvergleiche verschiedener Prozesse nur die reine verbrauchte Rechenzeit (CPU-Zeit) verwendet werden. Bei Verwendung der reale Laufzeit (Realzeit) können die Ergebnisse durch zufällige äußere Einflüsse variieren. Bei einem sequentiellen Prozeß ist die Realzeit immer höher als die CPU-Zeit. Benutzt ein Prozeß mehrere Prozessoren, so sagt die verbrauchte CPU-Zeit nichts mehr über seine Realzeit aus, da sie aus der Kumulation der CPU-Zeiten aller Prozessoren besteht. Eine Bestimmung der Gesamt-CPU-Zeit, wie sie für Vergleiche mit sequentiellen Prozessen notwendig ist, ist technisch aber nicht möglich. Die angegebenen Zeiten des parallelen Parsers sind daher Realzeiten, die auf einem gering ausgelastetem System ermittelt wurden. Genutzt wurden maximal vier CPUs, damit die Ausführung anderer Prozesse die Messungen nicht beeinflußte. Der reale *Speed-Up* ist daher vermutlich etwas besser als der in diesem Abschnitt angegebene. Die Ermittlung der Laufzeit des sequentiellen Parsers entspricht aber auch nicht der in der Theorie angegebenen Zeit T(1). Betriebssystem und Hardware sind für eine Parallelverarbeitung ausgelegt, bilden also nicht die optimale Plattform für den sequentiellen Parser. Eine der Theorie entsprechenden Berechnung der *Speed-Ups* ist daher praktisch ausgeschlossen.

Der erhoffte Zeitgewinn der Parallelisierung blieb aus. Die Abbildung 5.2 zeigt den Gewinn bzw. Verlust für die zehn Beispielsätze. Die Beamschwelle ist so gesetzt, daß kein Eintrag verworfen wird, damit ein Vergleich zu den späteren Versuchen ohne Beamschwelle möglich ist.

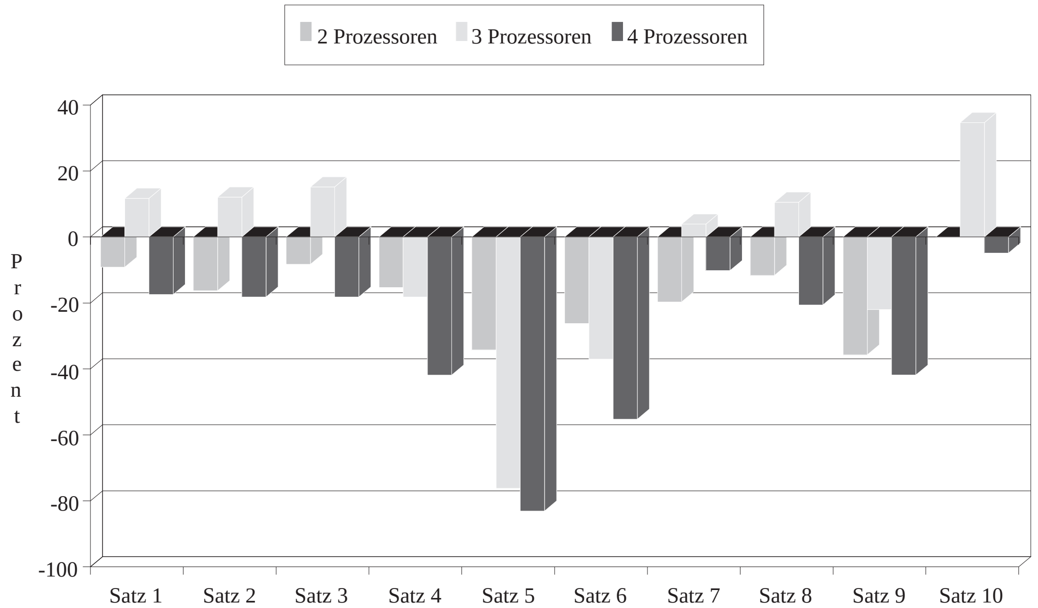


Abbildung 5.2 Zeitgewinn bzw. -verlust des parallelen Parsers in Prozent

Der durchschnittliche *Speed-Up* bei drei Prozessoren liegt bei 0,93. Zum Teil ist dieser Verlust algorithmischer Art. Eine Umwandlung von globalen in threadlokale Variablen, wie sie in Abschnitt 3.3 beschrieben wurde, ist in einigen hochfrequentierten Funktionen notwendig gewesen. Betrachtet man die Anzahl der ausgeführten Unifikationen in Tab. 5.1, so fällt auf, daß der Gewinn und Verlust beim *Speed-Up* von der Anzahl der Unifkationen beeinflußt wird.

	Satz 1	Satz 2	Satz 3	Satz 4	Satz 5	Satz 6	Satz 7	Satz 8	Satz 9	Satz 10
Unif.	3059	6008	3978	2249	408	1516	2824	4028	571	4979
Paare	20769	27814	19543	13745	2130	9418	17261	17891	5523	33604

Tabelle 5.1 Anzahl der ausgeführten Unifikationen und erzeugten Agendapaare pro Satz

Nicht alle Agendaeinträge führen zu einer Unifikation, weil jede Merkmalsstruktur mit einem Typ gekennzeichnet ist. Bevor zwei Strukturen unifiziert werden, werden ihre Typen verglichen. Sind die Typen unvereinbar, so wird die Unifikation erst gar nicht ausgeführt, da sie scheitern würde. Dieser Typcheck ist über einen sehr schnellen Bitvektorvergleich realisiert. Viele Agendaeinträge lassen sich also schon nach dem Typcheck verwerfen. Der Verwaltungsaufwand für diese Einträge übersteigt damit die eigentliche Arbeit. Je mehr Agendaaufträge zu einer echten Unifikation führen, desto besser ist der *Speed-Up*. Am schlechtesten ist er bei Lattices, die nicht erkannt wurden (Satz 5, 6 und 9). Scheitert das Weiterspannen aller aktiven Kanten zu einem sehr frühen Zeitpunkt, so werden im weiteren Verlauf der Analyse keine Unifikationen mehr durchgeführt. Ein anderer Punkt, der die Anzahl der Unifikationen beeinflußt, ist die Menge der im Wortgraphen enthaltenen syntaktisch korrekten Pfade. Je mehr Pfade möglich sind, desto mehr Unifikationen werden durchgeführt. Die Zahl der Pfade hängt zum einen von der Anzahl der Worthypothesen in der Lattice ab, zum anderen von der Ambiguität der verwendeten Grammatik.

Betrachtet man die Verteilung der Laufzeit der *Slaves* auf Wartezeiten und Rechenzeit (Abb. 5.3 bis 5.5), so läßt sich der Einbruch des *Speed-Ups* bei vier gegenüber drei Prozessoren erklären.

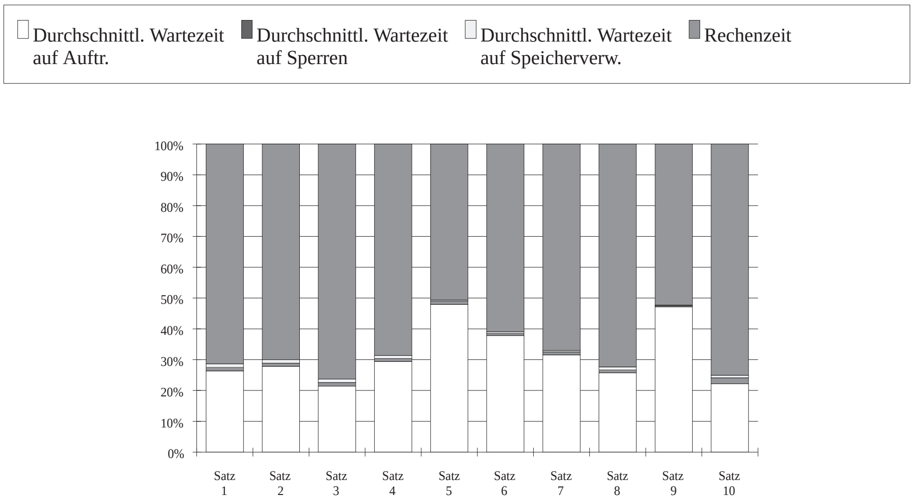


Abbildung 5.3 Durchschn. Aufteilung der Gesamtlaufzeit eines *Slaves* (2 Prozessoren)

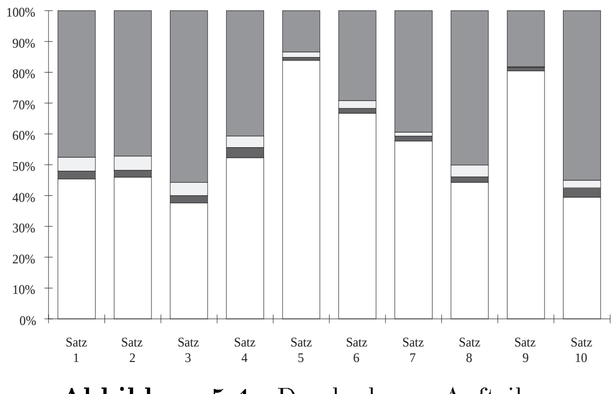


Abbildung 5.4 Durchschn. Aufteilung der Gesamtlaufzeit eines *Slaves* (3 Prozessoren)

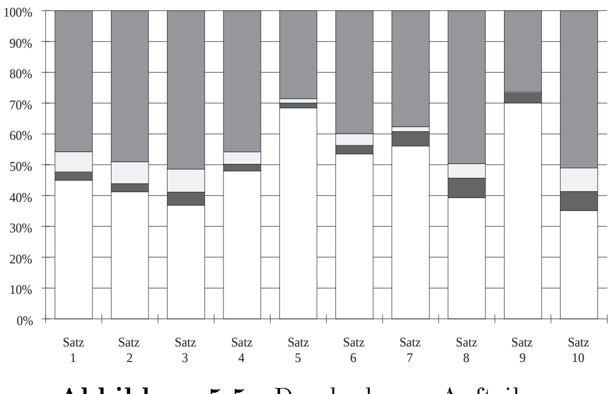


Abbildung 5.5 Durchschn. Aufteilung der Gesamtlaufzeit eines *Slaves* (4 Prozessoren)

Mit zunehmender Prozessorenzahl steigt erstens die Wartezeit jedes Prozessors auf Aufträge, da für gleichviele Aufträge mehr Prozessoren zur Verfügung stehen. Schon zwei Prozessoren sind durch ungünstige Aufgabenverteilung nicht immer ausgelastet. Die Parallelverarbeitung kann die algorithmischen Verluste in diesem Fall noch nicht ausgleichen. Bei drei Prozessoren sind diese Verluste zumindest bei den erkannten Sätzen aufgefangen. Die Auslastung jedes einzelnen *Slaves* sinkt aber schon. Des weiteren steigt mit jedem zusätzlichen Prozessor die Wahrscheinlichkeit für einen Zugriffskonflikt bei exklusiv genutzten Ressourcen. Dieses drückt sich in dem steigenden Anteil der Wartezeiten auf Sperren und Speicherverwaltung aus. Das Optimum zwischen Auftragsverteilung und Zugriffkonflikten liegt also bei drei Prozessoren.

Inwieweit der *Speed-up* durch den sequentiellen Anteil an der Gesamtrechenzeit, den der *Master* repräsentiert, beschränkt wird, zeigt Abbildung 5.6.

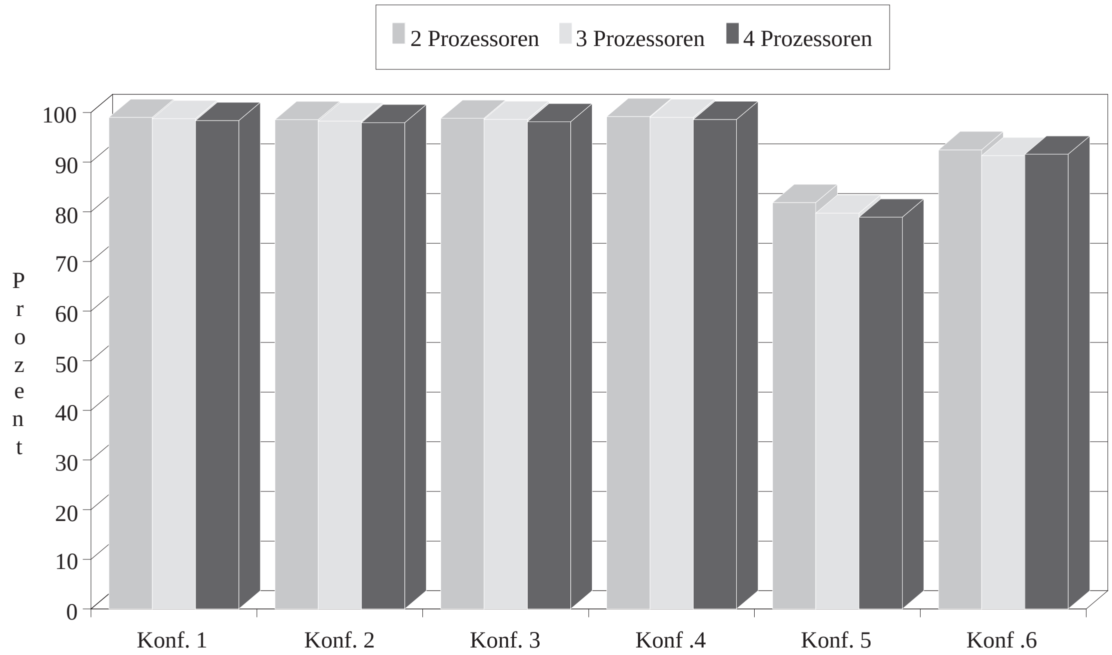


Abbildung 5.6 Wartezeit der *Masterthreads* in Prozent

Arbeitet der Parser nach den vorgestellten Schema (Konfiguration 5 und 6), so liegt der sequentielle Anteil des *Masters* zischen 10 und 20 Prozent. Bei Konfiguration fünf mit vier Prozessoren liegt er sogar über 20 Prozent. Mit dieser Konfiguration ist also bei optimaler Auslastung der *Slaves* nicht einmal ein *Speed-Up* um den Faktor fünf zu erzielen. Bei den Konfigurationenen 1 bis 4 wurden daher die Hypothesen parallel entgegengenommen. Auf eine globale Bestimmung der Beamschwelle wurde aufgrund der schon erwähnten aufwendigen Synchronisation verzichtet. Globales *Pruning* ist somit nicht mehr möglich. Der Anteil des *Masters* an der Gesamtlaufzeit liegt bei weniger als zwei Prozent. Er beschränkt den *Speed-Up* nach Amdahl damit auf das fünfzigfache. Das diese Geschwindigkeitssteigerungen noch nicht einmal annähernd erreicht wurden, liegt an den Wartezeiten der *Slaves*. Die Lastverteilung der Aufträge ist nicht optimal. Dieses Ungleichgewicht hat mehrere Gründe.

Der erste ist der schon beschriebene unterschiedliche Zeitbedarf für Agendapaare mit vereinbaren Typen und unvereinbaren Typen. Während ein *Slave* eine langsame Unifikation durchführt, können die anderen die Agenda evtl. ohne weitere Unifikation leeren.

Der zweite Grund ist die unzureichende Vorkompilierung der Grammatik. PROPOSE() muß zum Vorschlagen neuer Kanten alle Regeln der Grammatik durchsuchen. Bei einer unifikationsbasierten Grammatik ist dieses mit einem erheblichen Aufwand verbunden, da im Verlauf der Suche ebenfalls Unifikationen ausgeführt werden müssen. Die Implementation von PROPOSE() speichert zwar alle einmal zu einer Kategorie gefundenen Regeln in einer Hashtabelle ab, die Kategorien sind jedoch nicht statische Symbole, wie im kontextfreien Fall, sondern Merkmalsstrukturen, die im Verlauf der Analyse aufgebaut werden. Die linke Seite einer Regel kann daher in mehreren Ausprägungen vorhanden sein. Bevor also ein Agendapaar zu einem neuen Agendaeintrag, führt sind häufig mindestens zwei Unifikationen notwendig. Das Zeitverhältnis der Abarbeitung von Merkmalsstrukturen mit vereinbaren und unvereinbaren Typen verschiebt sich dadurch weiter zu ungunsten der Strukturen mit vereinbaren Typen. Der im Verhältnis zur Anzahl Unifikationen geringe *Speed-Up* bei Satz zwei läßt sich durch diesen Grund erklären.

Als dritter Punkt ist die Agendaverwaltung selbest zu nennen. Bei einer sortierten Agenda sind Entnahme- und Einfügeoperationen nicht mehr einfache Listenoperationen wie *push* und *pop* sondern selbst komplexe Funktionen. Die Wartezeit der Threads auf den exklusiven Zugriff auf die Agenda schränkt die Nebenläufigkeit weiter ein. Den Einfluß der Agendaverwaltung zeigt der Übergang von Konfiguration 2 zu Konfiguration 3 in Abbildung 5.7. Beide arbeiten ohne Beamschwellenberechnung. Der Rechenanteil des *Masters* ist daher sehr gering. Allein die längeren Wartezeiten bei einem Agendazugriff machen den geringen positiven *Speed-Up* der Konfiguration 2 wieder zunichte. Das Vorgehen, die Agenda ebenso wie die Chart in einen lokalen und einen globalen Teil aufzuteilen (Konfiguration 4 und 6), zeigte keine einheitliche Verbesserung. Aufträge wurden dabei nur dann auf die globale Agenda gelegt, wenn ein *Slave* auf neue Aufträge wartete. Bei dieser Aufteilung sammeln *Slaves* Aufträge auf ihren lokalen Agenden, solange in der globalen Agenda noch Aufträge vorhanden sind. Werden aber keine neuen Aufträge mehr produziert und ist die globale Agenda leer, so geht ein *Slave* nach dem letzten lokalen Agendaeintrag in den Wartezustand über, obwohl noch Aufträge auszuführen sind. Diese liegen aber unzugänglich in den lokalen Agenden anderer *Slaves*. Eine Verschärfung des *Load-Balancing* Problems ist die Folge. Abbildung 5.7 zeigt die prozentualen Zeitunterschiede bei den verschiedenen getesteten Konfigurationen.

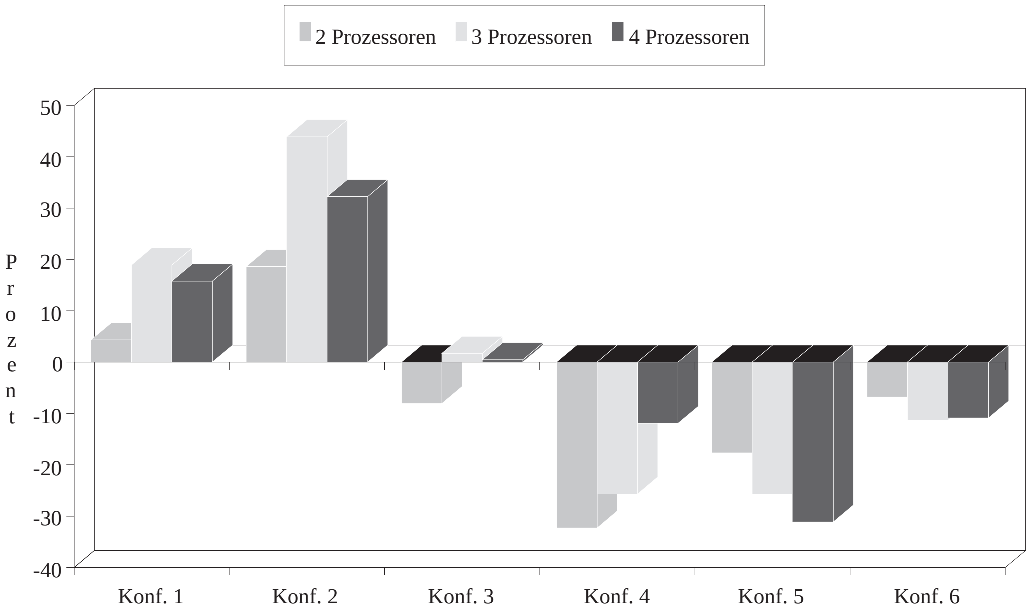


Abbildung 5.7 Durchschnittlicher Zeitgewinn und -verlust der getesteten Konfigurationen

Die Beschreibung der Konfigurationen erfolgt in Tabelle 5.2.

	Konfig. 1	Konfig. 2	Konfig. 3	Konfig. 4	Konfig. 5	Konfig 6
Redundanzcheck	lokal	global	global	global	global	global
Agendaverwaltung	Stack	Stack	sortiert	sortiert	sortiert	sortiert
Pruning möglich	nein	nein	nein	nein	ja	ja
Agendaart	global	global	global	lokal	global	lokal

Tabelle 5.2 Übersicht über die getesteten Konfigurationen

6
AUSBLICK

Die Ergebnisse bestätigen einen grundsätzliche Zusammenhang bei der Parallelisierung von Programmen. Je effizienter das sequentielle Programm ist, desto schwieriger ist ein Geschwindigkeitsgewinn in der Parallelversion zu erzielen. Vor allem Optimierungen der sequentiellen Version, die sich nicht ohne weiteres parallelisieren lassen, machen Parallelisierungsgewinne schnell zunichte. Weber hat z.B. für die sequentielle Version einen inkrementellen *Time-Map* implementiert (siehe [Web95, S. 88]), der weitere Unifikationen spart. Bei diesem Verfahren werden Kanten, die aus Worthypothesen entstehen, die sich nur im Endknoten von schon bekannten Hypothesen unterscheiden, vererbt. Im parallelen Fall muß diese Operation der *Master* durchführen. Der parallele Anteil des Parsers wird also zugunsten des sequentiellen Anteils kleiner. Der *Speed-Up* sinkt.

Andererseits gibt es Optimierungen, von denen der parallele Parser mehr profitiert als der sequentielle. Eine komplette Vorkompilierung der Grammatik, wie sie in [Web95, S. 43] beschrieben ist, führt in beiden Fällen zu einer Zeitersparnis bei der PROPOSE()-Funktion. Bei dem parallelen Parser würde diese Vorkompilierung aber gleichzeitig die Wartezeit auf neue Aufträge verkürzen. Der parallele Parser profitiert also doppelt von dieser Optimierung.

Trotz dieser Optimierung wächst die Laufzeit des Parsers exponentiell mit der Menge der Worthypothesen und der Größe der Grammatik. Bei neuen Tests mit dem sequentiellen Parser, bei dem von der kleinen Beispieldomäne mit sauber gesprochenen Sätzen zu spontanter Sprache übergegangen wurde, steigt die Laufzeit vom Sekunden- in den Stundenbereich. Die Grammatik zur spontanen Sprache hat vor allem eine hohes Maß an Ambiguität, von der die parallele Version des Parsers besonders profitieren würde. Bis zu 100 Lesarten einer Äußerung sind dabei beobachtet worden. Die dabei aufgebauten Merkmalsstrukturen sind wesentlich größer als bei den Beispielsätzen, da sie schon semantische Informationen enthalten. Eine einzelne Unifikation dauert damit länger. Außerdem steigt die Menge der Worthypothesen bei Spontansprache erheblich an, da die Worterkennung durch ungünstigere Randbedingungen schlechter ist. Ein Einsatz des parallelen Parsers scheint aufgrund der genannten Aspekte in diesem Bereich erfolgversprechend.

Unabhängig vom Parser ist eine bessere Parallelisierung des LISP-Systems notwendig. Viele threadspezifische Daten, die als LISP-Felder implementiert sind, ließen sich bei einer anderen *special-variable* Semantik auf threadspezifische Daten auf der C-Ebene abbilden. (siehe dazu Abschnitt 3.3). Damit ist ein wesentlich schnellerer Zugriff möglich. Außerdem ist eine Alloziierungsfunktion notwendig, die die Speicheranforderungen mehrerer Threads gleichzeitig erfüllen kann. Je weniger Wartezeit auf Aufträge der Parser aufweist, desto wichtiger ist eine konfliktfreie Speicherverwaltung.

Als Konsequenz für die Konstruktion eines parallelen Parsers sind zwei Erkenntnisse zu nennen. Inkrementelles Parsing bietet grundsätzlich nicht soviel logische Parallelität wie nicht-inkrementelles, da an jedem Knoten mindestens eine Synchronisation notwendig ist. Damit ist festgelegt, daß alle Threads auf den letzten arbeitenden Thread warten müssen. Diese Wartezeit muß durch möglichst gleichgroße Teilaufgaben minimiert werden. Eine Alternative ist die Abschwächung der Forderung, erst dann zum nächsten Knoten überzugehen, wenn die Agenda für den aktuellen Knoten vollständig abgearbeitet ist. Damit würde jeder Thread für sich inkrementell arbeiten. Zum Dekoder würde sich das System zwar weiterhin wie ein links-rechts-inkrementelles System verhalten, ein Semantikmodul müßte jedoch damit rechnen, Ergebnisse zu bekommen, die vor dem aktuellen letzten Knoten liegen. Für den Parser würde diese Abschwächung außerdem bedeuten, daß ein globales Pruning nicht mehr möglich ist. Eine Konsequenz der ungleichen Lastverteilung ist die Aufspaltung des Parsers in einen Unifikator mit mehreren Threads und einen kontextfreien Chart-Parser. Der Chart-Parser übernimmt die Aufgabe des Masters, der den kontextfreien Teil der Regeln auswertet. Immer wenn eine kontextfreie Regel erfolgreich erweitert wird, führt dieses zu einem Unifikationsauftrag mit den annotierten Merkmalsstrukturen. Dieser wird asynchron von einem der Threads des Unifikators ausgeführt. Damit wird eine Synchronisation nur für zeitintensive Aktionen notwendig.

Diese Aspekte zeigen, daß der Einsatz eines Parallelrechners bei der Verarbeitung gesprochener Sprache nicht grundsätzlich falsch ist, auch wenn die erzielten Ergebnisse keinen positiven Eindruck vermitteln. Dabei ist vor allem wichtig, daß schon bei der Konzeption auf die Parallelverarbeitung eingegangen wird und nicht bestehende Programme als Ausgangspunkt einer Parallelisierung dienen. Dieses kann aber nur durch ein Umdenken bei der Programmentwicklung geschehen. Es bleibt zu hoffen, daß sich mit der zunehmenden Verbreitung von Multiprozessoren auch Methoden der Softwareentwicklung etablieren, die dem Rechnung tragen.

A

MULTITHREAD-INTERFACE IN LISP

Die folgenden LISP-Makros sind in dem *Package MULTITHREAD* zu finden. Über den Eintrag *multithreading* in der *Feature*-Liste läßt sich das Interface aktivieren. Dieses muß gesetzt sein, bevor die Programmdatei *multithread.cl* geladen wird. Nach einer Deaktivierung ist das komplette Programm neu zu übersetzen. Die im folgenden mit *name* gekennzeichneten Namensbezeichner müssen Symbole sein, die keine mit der C-Syntax unverträgliche Zeichen enthalten.

Funktion	Parameter	Beschreibung
declare-lock	*name*	Deklariert eine Sperre mit dem Namen *name*. Diese Funktion darf nur auf dem Toplevel benutzt werden.
init-lock	*name comment*	Initialisiert eine Sperre mit dem Namen *name*. *comment* ist ein String, der als Kommentar benutzt wird.
get-lock	*name*	Fordert die Sperre *name* an. Hält ein anderer Thread diese Sperre, so blockiert der Aufrufer, bis die Sperre freigegeben wird.
release-lock	*name*	Gibt die Sperre *name* frei. Falls andere Threads auf diese Sperre warten, wird sie einem der Wartenden zugeteilt. Die Auswahl ist aber nicht festgelegt.
new-thread	*init-fn parameter*	Erzeugt einen neuen Thread. *init-fn* ist die Funktion, die als erstes von dem Thread ausgeführt wird. Der Thread terminiert mit dem Ende der Funktion. *init-fn* muß eine Funktion sein, die genau einen Parameter erwartet. Dieser wird an *parameter* gebunden.
wait-for-threads		Der aufrufende Thread wartet auf das Ende aller anderen Threads.
thread-id		Liefert eine eindeutige Kennung des Threads zurück. Diese sollte aber nur für die Funktionen des *Multithreading-Interface* genutzt werden, da sie implementationsabhängig ist.
thread-kill	*id*	Beendet den Thread mit der Kennung *id*.

Tabelle A.1 Makros des MT-Interface

Funktion	Parameter	Beschreibung
process-recursive-with-lock	*lock-name body*	Wertet den Block *body* unter dem Schutz der Sperre *lock-name* aus. Trifft der Thread während der Auswertung auf einen weiteren *process-with-recursive-lock* mit derselben Sperre, so blockiert er nicht. (Vorsicht! noch nicht ausreichend getestet.)
declare-condition	*name*	Definiert ein Objekt zur Verwaltung von Zustandsänderungen (Conditions) mit dem Namen *name*. Dieser Befehl darf nur auf dem Toplevel benutzt werden.
init-condition	*name*	Initialisiert ein *Condition*-Objekt.
wait-on-condition	*name function comment*	Wartet an der *Condition*, bis das Prädikat *function* das Eintreten des gewünschten Zustandes anzeigt. *comment* wird als Kommentar verwendet.
read-condition	*name form*	Ändert den Zustand, der von dem Objekt *name* verwaltet wird, durch die Auswertung von *form*. Auf Zustandsänderungen von *name* wartende Threads werden evtl. zur Überprüfung ihres Prädikats aufgefordert.
change-condition	*name form*	Ändert den Zustand, der von dem Objekt *name* verwaltet wird, durch die Auswertung von *form*. Mindestens ein auf eine Zustandsänderung von *name* wartender Thread wird zur Überprüfung seines Prädikats aufgefordert.
broadcast-change-condition	*name form*	Ändert den Zustand, der von dem Objekt *name* verwaltet wird, durch die Auswertung von *form*. Alle auf Zustandsänderungen von *name* wartende Threads werden zur Überprüfung ihres Prädikats aufgefordert.

Tabelle A.1 - *Fortsetzung* - Makros des MT-Interface

Anmerkung: Zwischen der Aufforderung, ein Prädikat zu überprüfen und dem Aufrufen des Prädikat kann ein unbestimmt langer Zeitraum liegen. Eine zwischenzeitlich vorgenommene erneute Änderung kann dazu führen, daß das Prädikat den Wert *nil* liefert, obwohl es zum Zeitpunkt der Aufforderung erfüllt war.

B

TESTBEISPIELE FÜR PARALLELES LISP

Die folgende zwei Beispielprogramme zeigen Grenzen und Fähigkeiten des verwendeten parallelen LISP. Das erste Programm löst das Problem, auf einem Schachbrett der Größe n eine Stellung für n Damen zu finden, in der keine Dame geschlagen werden kann. Dieses ist eine Verallgemeinerung des 8-Damen-Problems. Implementiert ist es durch einen Algorithmus, der weder Rekursionen noch Speicheranforderungen während der Lösungssuche aufweist. Die Parallelität in den Tests beschränkt sich auf eine mehrfache Ausführung der Lösungsfunktion. Eine Parallelisierung des Algorithmus wurde nicht vorgenommen. Damit sind programmtechnisch ideale Bedingungen für einen maximalen *Speed-Up* gegeben. Weder algorithmische noch Synchronisationsverluste können auftreten. Auch das *Load-balancing* Problem wird vermieden. Eine problembedingte Reduzierung des *Speed-Ups* kann nur durch Threaderzeugung und -terminierung auftreten. Weitere Verluste können durch das LISP-System, das Betriebssystem oder die Hardware hervorgerufen werden. Die Abbildung B.1 zeigt den erzielten *Speed-up* bei einem vierfachen Aufruf der Lösungsfunktion und dem Einsatz von vier Prozessoren.

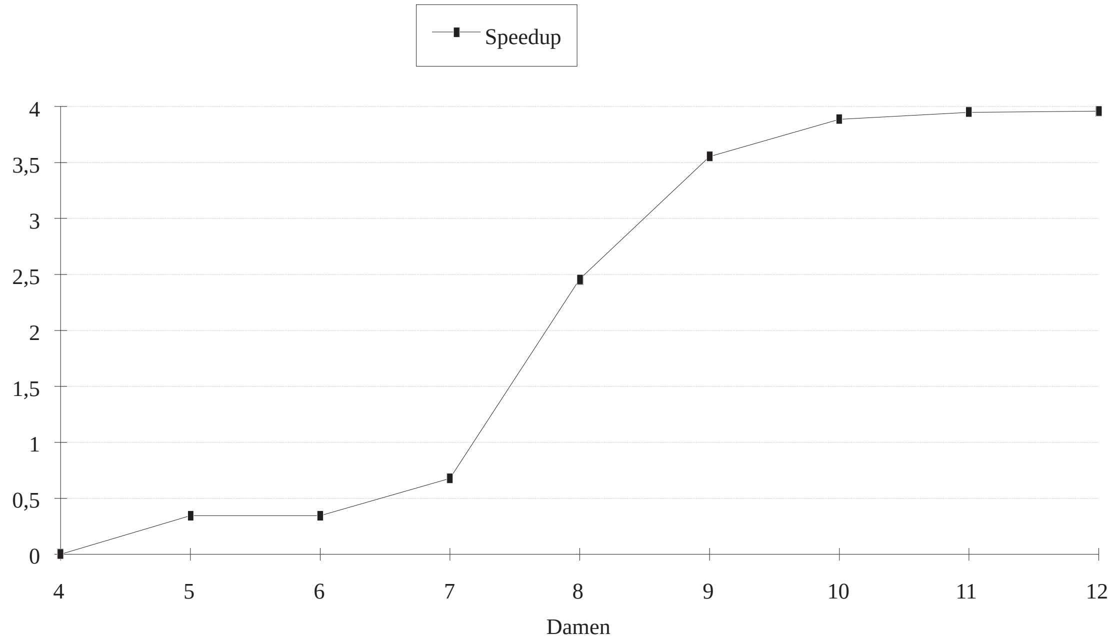


Abbildung B.1 *Speed-Up* bei dem Damenproblem

Messungen mit einer Schachbrettgröße von drei oder kleiner waren nicht möglich, da das zeitliche Auflösungsvermögen der Meßfunktion zu grob ist. Der theoretische *Speed-Up* von vier wird bei großen Spielbrettern erreicht. Prinzipelle Einschränkungen des Systems bestehen also nicht. Die ungünstigeren *Speed-Ups* kleinerer Bretter zeigen aber den Einfluß der Threaderzeugungs- und Terminierungszeiten. Erst bei einem Schachbrett der Größe acht wird dieser Verlust wieder ausgeglichen.

Das zweite Beispiel demonstriert den Einfluß der Alloziierungsfunktion auf den *Speed-Up*. Es berechnet mehrfach die Fakultätsfunktion. Die Parallelität beschränkt sich wieder auf die mehrfache Berechnung desselben Parameters. Die Implementierung ist ebenfalls rekursionsfrei, fordert aber ständig neue Speicherbereiche an. Der exklusive Zugriff auf die Alloziierungsfunktion bildet damit den Engpaß des Programms.

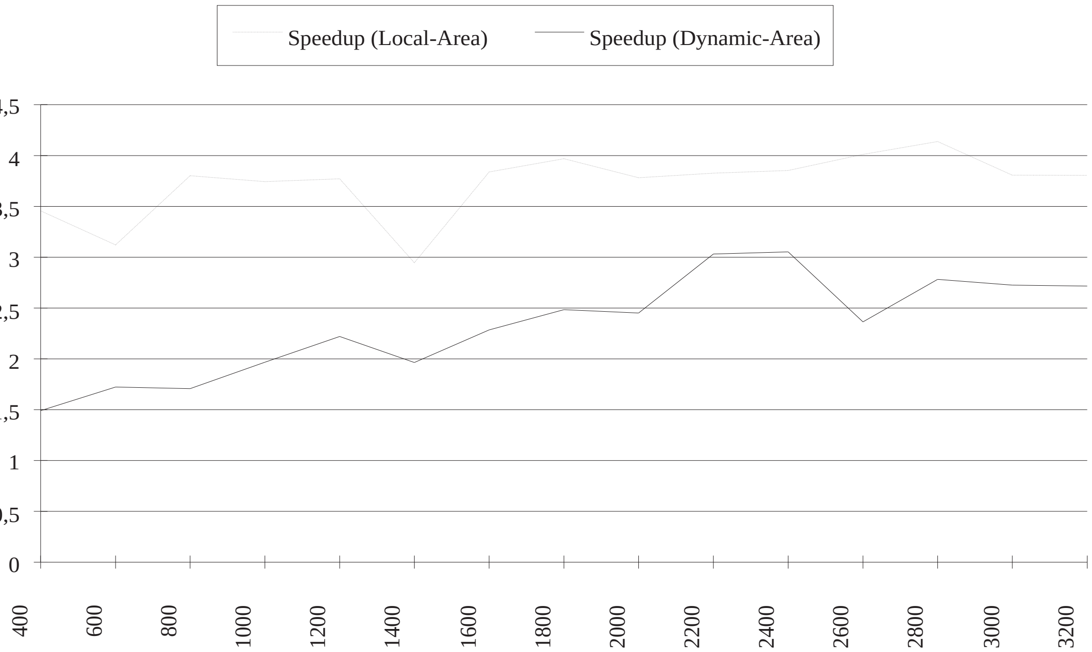


Abbildung B.2 *Speed-Up* bei vierfacher Ausführung der Fakultätsfunktion mit vier Prozessoren

Wird aus dem normalen LISP-Speicher (Dynamic-Area)[1]alloziiert, steigt der *Speed-Up* auch bei großen Parametern, bei denen die Threadverwaltungzeit keine Rolle mehr spielt, nicht über drei. Die Wartezeiten auf angeforderte Speicherbereiche sind hierfür verantwortlich. Alloziiert man Speicher aus den *Local-Areas*, so ergibt sich nicht ganz die erwartete Leistungssteigerung. Das Beispiel weist im Gegensatz zum ersten keine so hohe Lokalität bei den verwendeten Hauptspeicherseiten auf. Die Trefferrate des CPU-Cache sinkt, die Zugriffs- und Synchronisationszeiten für die benutzten Seiten steigen. Der positive „Ausrutscher“ bei der Berechnung der Fakultät von 2800 ist wohl eher auf die problematische Zeitmessung[2], als auf einen echten superlinearen *Speed-Up* zurückzuführen. Es zeigt sich, daß selbst bei einer optimalen Parallelisierung eines LISP-Programms mit der aktuellen LISP-Implementation nur eine Leistungssteigerung um den Faktor drei zu erzielen ist, wenn viele Objekte erzeugt werden.

[1] siehe Speicherverwaltung in Kap. 3.2

[2] siehe Kap. 5.4

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LITERATUR

[Amd67] Amdahl, G. Validity of the Single Processor Approach to Achieve Large Scale Computing Capabilities. In *Proceedings of the Conference of AFIPS*, Thompson Books, Washington D.C., 1967.

[Amt90] Amtrup, Jan. POP. Ein Paralleler Objektorientierter Chart-Parser. Studienarbeit, Universität Hamburg, FB Informatik, AB NatS, Juli 1990.

[Amt92] Amtrup, Jan. Parallele Strukturanalyse natürlicher Sprache mit Transputern. Diplomarbeit, Universität Hamburg, FB Informatik, AB NatS, März 1992.

[Bac89] Backofen, Rolf. Integration von Funktionen, Relationen und Typen beim Sprachentwurf. Diplomarbeit, Universität Erlangen- Nürnberg, 1989.

[Bre92] Brehm, Jürgen. *Parallele lineare Algebra*. Deutscher Universitäts-Verlag, Wiesbaden, 1992.

[CCL90] Chien, Lee-Feng; Chen, K.J.; Lee, Lin-Shan. An Augmented Chart Data Structure with Efficient Word Lattice Parsing Scheme in Speech Recognition Applications. In *Proceedings of COLING*, pp. 60–65, 1990.

[Dei84] Deitel, H. M. *An Introduction on Operating Systems*. Addison-Wesley Publishing Company, Reading, Mass., 1984.

[Dij68] Dijkstra, E.W. *Co-Operating Sequential Processes*, pp. 43–112. Academic Press, New York, 1968.

[Fly66] Flynn, M.J. Very-High-Speed Computing Systems. In *Proceedings of the 54th IEEE Conference*, 1966.

[Fra92] Franz Inc. *Allegro CL User Guide*, 1992.

[GC88] Grishman, R.; Chitrao, M. Evaluation of a parallel chart parser. In *Second Conference on Applied Natural Language Processing*, pp. 71–76, Association for Computational Linguistics, Austin, Texas, Februar 1988.

[Gör88] Görz, Günther. *Strukturanalyse natürlicher Sprache*. Addison Wesley, Bonn, 1988.

[Haa87] Haas, Andrew. Parallel Parsing for Unification Grammars. In *Proceedings of the International Joint Conference on Artifical Intelligence*, pp. 615–618, Mailand, 1987.

[Hal89] Halstead, Jr., Robert H. *Design Requirements of Concurrent Lisp Machines*, Kapitel 3, pp. 69–105. McGraw-Hill Series in Supercomputing and Parallel Processing. McGraw-Hill Publishing Company, New York, 1989.

[HL85a] Halstead, Jr., Robert H.; Loaiza, Juan R. Exception Handling in Multilisp. In *Proceedings of the International Conference on Parallel Processing*, pp. 822–830, August 1985.

[HL85b] Halstead, Jr., Robert H.; Loiza, Juan R. Multilisp: A Language of Concurrent Symbolic Computing. In *ACM Transaction on Programming Languages and Systems, Vol.7*, pp. 501–538, Oktober 1985.

[Hof91] Hofmann, Fridolin. *Betriebssysteme: Grundkonzepte und Modellvorstellungen*. B.G. Teubner, Stuttgart, 1991.

[Hoß83] Hoßfeld, F. *Parallele Algorithmen*. Springer-Verlag, Berlin Heidelberg, 1983.

[Jac90] Jacob, George K. What price, parallel Lisp? In *Europ. Workshop: High Performance and Parallel Computing in LISP*, pp. 1–12, Londen, November 1990.

[Kay73] Kay, Martin. The MIMD System. In Rustin, R, Hrsg. *Natural Language Proceesing*, pp. 155–188, Algorithmic Press, New York, 1973.

[Kes90] Kesseler, Marcus P. Entwurf und Implementierung eines verteilten Scheme-Lisp Systems für Transputernetzwerke. Diplomarbeit, Universität Erlangen-Nürnberg, Erlangen, Oktober 1990.

[Mar82] Marr, D. *Vision. A Computational Investigation into a Human Representation and Processing of Visual Information*. Freeman, New York, 1982.

[Mat88] Matsumoto, Yuij. A Parallel Parsing System for Natural Language Analysis. In *Third International Conference on Logic Programming*, Nr. 225 in Lecture Notes in Computer Science, pp. 396–409, Berlin Heidelberg, Juli 1988. Imperial College of Science and Technology, Springer Verlag.

[MWT80] Marslen-Wilson, W. D.; Tyler, L. K. *The temporal structure of spoken language understanding*. *Cognition*, 8:1–71, 1980.

[Nij89] Nijholt, A. Parallel parsing strategies in natural language. In *Proceedings of the International Workshop on Parsing Technologies*, pp. 240–253, Carnegie Mellon University, Pittsburgh, USA, August 1989.

[PKB+91] Powell, M.L.; Kleiman, S.R.; Barton, S.; Shah, D.; Stein, D.; Weeks, M. SunOS Multi-Thread Architecture. Technical report, SUN Microsystems, Inc, Mountain View, California, 1991.

[PW80] Pereira, F.C.N.; Warren, David H. D. *Definite Clause Grammars for Language Analysis – A Survey of the Formalism and a Comparison with Augmented Transition Networks*. *Artificial Intelligence*, 13:231–278, 1980.

[SH86] Steele, Guy L.; Hillis, W. Daniel. Connection Machine Lisp: Fine Grain Parallel Symbolic Processing. In *ACM Conference on LISP*, pp. 279–297, 1986.

[Shi86] Shieber, Stuart M. *An Introduction to Unification-based Approaches to Grammar*. CSLI, Lecture Notes, Stanford, 1986.

[SUN92] SUN Microsystems, Inc, Mountain View, California. *SPARCCenter 2000 Architecture and Implementation*, 1992.

[SUN94] SUN Microsystems, Inc, Mountain View, California. *Multithreading Programming Guide*, 1994.

[Tho84] Thompson, H.S. Chart parsing for loosely coupled parallel systems. In *Proceedings of the International Workshop on Parsing Technologies*, pp. 320–328, Pittsburgh, USA, August 1984.

[TR84] Thompson, H.; Ritchie, G. Implementing natural language parsers. In O'Shea, T.; Eisenstadt, M., Hrsg. *Artificial Intelligence — Tools, Techniques and Applications*, pp. 245–300. Harper and Row, London, 1984.

[Wal95] Waldschmidt, Klaus, Hrsg. *Parallelrechner: Architekturen - Systeme - Werkzeuge*. B.G. Teubner, Stuttgart, 1995.

[Web95] Weber, Hans H. *LR-inkrementelles probabilistisches Chartparsing von Worthypothesenmengen mit Unifikationsgrammatiken: Eine enge Kopplung von Suche und Analyse*. Dissertation, Universität Hamburg, FB Informatik, 1995.

[Wil92] Wilson, Paul R. Uniprocessor Garbage Collection Techniques. In *International Workshop on Memory Management*, Band Lecture Notes in Computer Science no. 637, pp. 1–42. Springer Verlag, September 1992.

[Win83] Winograd, T. *Language as a Cognitive Process. Volume I: Syntax*. Addison-Wesley Publishing Company, Reading, Mass., 1983.

[Wir92] Wirén, M. *Studies in Incremental Natural-Language Analysis*. Nr. 292 Dissertation in Linköping Studies in Science and Technology. Linköping University, 1992.

[WS93] Weng, Fulaing; Stolcke, Andreas. *Partitioning Grammars and composing Parsers*. SRI. Menlo Park, 1993

[YO89] Yonezawa, Akinori; Ohsawa, Ichiro. *Object-Oriented Parallel Parsing for Context-Free Grammars*, pp. 219–222. MIT-Press, 1989.