Parallel Natural Language Parsing: From Analysis to Speedup

Parallel Natural Language Parsing: From Analysis to Speedup

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.ir. K.F. Wakker,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op dinsdag 6 november 2001 om 10:30 uur

door Marcellus Paulus VAN LOHUIZEN
informatica ingenieur
geboren te Alphen aan den Rijn.

Dit proefschrift is goedgekeurd door de promotoren:
Prof.dr.ir. H.J. Sips
Prof.dr.ir. A. Nijholt

Toegevoegd promotor:
Dr.ir. R. Sommerhalder

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof.dr.ir. H.J. Sips,	Technische Universiteit Delft, promotor
Prof.dr.ir. A. Nijholt ,	Universiteit Twente, promotor
Dr.ir. R. Sommerhalder,	Technische Universiteit Delft, toegevoegd promotor
Prof.dr.ir. J.L.G. Dietz,	Technische Universiteit Delft
Prof.dr. H.J. van den Herik,	Universiteit Maastricht
Prof.dr. F.J. Peters,	Universiteit Leiden
Dr. J.A. Carroll,	University of Sussex, Groot-Brittannië

Published and distributed by:

Marcel P. van Lohuizen
ISBN 90-9015281-4

Keywords: parallel parsing, natural language

Copyright © 2001 by Marcel P. van Lohuizen

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage retrieval system, without written permission of the author.

E-mail author: mp@vanlohuizen.com

Preface

The research presented in this thesis done in the context of the "Parallel Natural Language Interfaces" subproject of the IMPACT project. The IMPACT project was headed by the ING bank and founded by the organization for High Performance Computing and Networking (HPCN), itself partly funded by NWO. The goal of this subproject was to improve the performance of natural language interfaces. Considering the context of the research, there was a strong focus on achieving speedup by means of parallel processing. Although the research goals aim at improving the performance of natural language interfaces in general, the research in this thesis focuses on parallel parsing. The reason for this was that the parsing component of the Deltra system, which initially was the target platform of our research, was by far the most computationally expensive component. The parsing component is often the most computationally expensive component for other natural language processing applications as well. To broaden the scope of our research, we also investigated a parser for a different grammar (LinGO) at a later stage of the project.

Although a dissertation often appears to be an individual achievement, there are a number of people to whom I am indebted. First and foremost, I would like to thank my promotors Henk Sips and Anton Nijholt and added promotor Ruud Sommerhalder for the guidance and support they have given me. I am especially grateful to Ruud Sommerhalder for his involvement with the successful completion of my dissertation.

Also, the contents of this thesis would not have been what it is had I not had the right people around me. In this respect, I would like to thank several people. Firstly, I would like to thank Job Honig, for the many discussions often related to natural language parsing, Stephan Oepen, for providing me with the opportunity to get on track with the LinGO grammar, and Ulrich Callmeier for the help and support with the porting effort to make my parser compatible with the LinGO grammar. Secondly, I would like to thank all the people that enabled me to run my experiments on the right machinery: Jan Hol, for allowing me to use the 8 processor UltraSparc of the Faculty of Aerospace Engineering, and Takashi Ninomiya and Makino Takaki of the Department of Information Science at the University of Tokyo, for running experiments on their 64 processor UltraSparc. In computer science research, often little can be done without the availability of the right machinery.

Thirdly, I would like to thank all the colleagues that may not have been directly involved with my research, but have nevertheless given a great deal of support. Here an attempt to name all of them: my office room mates throughout the years: Leon Aronson, Mathijs de Weerdt, Roman van der Krogt, Paul Dechering, and David Lindeijer, which helped a great deal with all kinds of questions and who hardly ever told me to shut up when I was talking too much, other fellow Ph.D. students: Anca Bucur, Jeroen Valk, Ihor Kuz, Johan Pouwelse, Jonne Zutt and Ruud de Rooij, the system administrators: Paulo Anita, Onno Roep, and Hub Engelen, the secretaries: Toos Brussee and Coby Bouwer, who were of great support, and all the other colleagues of the department I spent a great deal of time with and who were always willing to help out: André Bos, Leo Breebaart, Dick Epema, Jan Heijnsdijk, Frits

Kuijlman, Koen Langendoen, Kees van Reeuwijk, Hans Tonino and Cees Wittenveen.

Finally, I would like to thank my mother for making the necessary arrangements in the final phase of my promotion, for putting up with me in the last year when I was overly consumed with my work and not always the nicest person, and for always being there when I needed her.

Mountain View, CA, September 2001 Marcel P. van Lohuizen

Contents

Preface v

1 Introduction 1

1.1 Natural Language Processing . . . 1
1.2 Focus of Research . . . 3
1.3 Context . . . 4
1.4 Organization of the Thesis . . . 5

2 Natural Language Parsing 7

2.1 Parsing Schemata . . . 7
2.1.1 Parsing Schemata . . . 8
2.1.2 Mappings between Parsing Schemata . . . 11
2.1.3 Common Parsing Schemata . . . 12
2.1.4 Parsing Schemata for Parallel Parsing . . . 16
2.2 Unification-Based Grammars . . . 17
2.2.1 Unification of Typed Feature Structures . . . 18
2.2.2 Unification-based Grammars . . . 21
2.2.3 Algorithms for Unification . . . 22
2.2.4 Parallel Unification . . . 27
2.3 Tabular Parsing . . . 28
2.3.1 Chart Parsing . . . 29
2.3.2 Optimizations . . . 30
2.3.3 Parallel Chart Parsing . . . 31
2.4 Existing Parallel Natural Language Parsers . . . 32

3 Analysis and Optimization of Parallel Computations 35

3.1 Parallel Computing . . . 35
3.1.1 Parallel Models . . . 35
3.1.2 Performance of Parallel Computations . . . 37
3.2 Scheduling . . . 37
3.2.1 Bounds for Optimal Schedulings . . . 38
3.2.2 Scheduling Algorithms . . . 38
3.2.3 Work-first Principle . . . 40

3.3 Domain Decomposition 41
3.3.1 Introduction 41
3.3.2 Partitioning Algorithms 42
3.4 Caching Principles 43
3.4.1 Terminology 43
3.4.2 Techniques to Improve Cache Utilization 45

4 Parallelism in Parsing Computations 47
4.1 Introduction 47
4.2 Parsing as Deduction 47
4.2.1 Definitions 48
4.2.2 Strict Parsing Systems 49
4.3 Task-Graph Analysis 50
4.3.1 Task Dependencies 50
4.3.2 Task Graphs 53
4.3.3 Metrics on the Task Graph 54
4.3.4 Results 55
4.4 Parallelism and Parsing Schemata 56
4.4.1 Justification Graph as Task Graph 58
4.4.2 Base Parsing Schema 58
4.4.3 Measurements 59

5 Communication in Parallel Parsing 63
5.1 Introduction 63
5.2 Communication Model 63
5.3 Grouping Heuristics 65
5.3.1 Reducing the Upper bound of Communication 65
5.3.2 Grouping Heuristics 66
5.4 Evaluation 67

6 Efficient Thread-safe Unification 71
6.1 Introduction 71
6.2 Separating Scratch Fields 72
6.3 The Indexing Technique 74
6.3.1 The Indexing Algorithm 75
6.3.2 Structure Sharing 78
6.3.3 Other Optimizations 80
6.4 Performance Comparison 81

6.4.1 Results for LinGO . . . 81
6.4.2 Results for Deltra . . . 83
6.5 Comparison to other Algorithms . . . 83
6.6 Conclusions and Future Directions . . . 85

7 Design and Implementation of a Parallel Parser 87
7.1 Introduction . . . 87
7.2 Scheduling . . . 88
7.2.1 Data Structures . . . 90
7.2.2 Scheduling Algorithm . . . 90
7.2.3 Preventing Duplicate Matches . . . 92
7.2.4 Work Stealing . . . 93
7.2.5 Termination . . . 95
7.3 Implementation . . . 96
7.3.1 Overall architecture . . . 97
7.3.2 Parsing Process . . . 98
7.4 Theoretical Analysis . . . 98
7.5 Empirical Analysis . . . 100

8 Optimizing Cache Performance 105
8.1 Introduction . . . 105
8.2 Model . . . 107
8.2.1 Memory Layout . . . 107
8.2.2 Cache Miss Equations . . . 108
8.2.3 Choosing the Right Value for b . . . 110
8.3 Optimizing Spatial Reuse . . . 111
8.4 Blocking Strategies . . . 112
8.4.1 Simple Blocking . . . 112
8.4.2 Categorized Blocking . . . 113
8.5 Results . . . 113

9 Conclusions and Suggestions for Future Research 117

References 121

Summary 131

Summary in Dutch 137

About the Author 143

Index 145

Chapter 1

Introduction

The most natural way of communication for humans is, in most cases, the use of natural language. The field of natural language processing aims at giving computers the ability to process or understand human language for any of a wide variety of applications. Applications of natural language processing that aim at coming to a full understanding of a given sentence or text are typically computationally intensive. The trade off between linguistic accuracy and computational performance can therefore be an important issue. In this thesis, we will investigate the possibilities for improving the performance of natural language processing by using modern day multiprocessors.

1.1 Natural Language Processing

In recent years, the field of natural language processing has gradually found its way from theory to specific, often large-scale, applications. Just about a decade ago, most of the research focussed on, for example, knowledge representation and small-scale research systems. Although theoretical issues remain an important topic of research, a lot of today's research focuses on large-scale applications of natural language processing for real-life problems. Some of the major forces prompting this shift were EU and US government sponsoring of research on specific applications, simple market-driven demand, and the overall Zeitgeist itself (Wilks, 1996), (edi, 1995).

Natural language processing has many applications, including machine translation, for the automated translation of written or spoken language, natural language interfaces, to provide simple interfaces to otherwise complicated software or machinery, information extraction, for the automated extraction of information from texts, and information retrieval, where the goal is to automatically retrieve a selection of documents that contain the relevant information according to some query.

All of these applications have in common that they incorporate some knowledge about one or more natural languages. The level of sophistication of the linguistic knowledge they incorporate, however, can differ considerably. Some applications are limited to identifying word categories, often based on statistical methods. Most modern applications for information retrieval, for example, use word stemming as the sole source of linguistic knowledge. Other applications use more sophisticated models, which model the structure, or syntax, and meaning, or semantics, of the language. Such information is often specified in a grammar formalism that is specified within the context of some linguistic theory. These more sophisticated approaches to processing of natural language can be very computationally expensive. In addition, the development of a broad-coverage grammar that covers a significant number

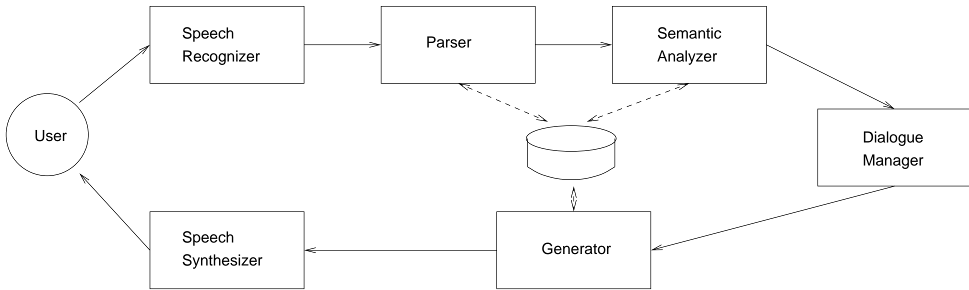


Figure 1.1 Outline of a common architecture for a speech-based natural language processor.

of aspects of a language is very time consuming. One approach to simplify the linguistic analysis is to limit the coverage of such systems to a specific domain of discourse. Another approach is to perform only a partial analysis of the syntax of sentences. Often, statistical methods are used in determining the role of a certain word or constituent.

Sometimes a partial analysis of the syntax simply suffices to reach the level of semantic accuracy that is required for the application (Cowie and Lehnert, 1996). Many applications for information extraction, for example, follow this approach. For many applications, however, it seems that full syntactic and semantical analysis is a necessity. In general, there seems to be a trend to move towards more complex analysis as machines with more computing power and more memory become available.

For applications that aim at full linguistic analysis, there seems to be a consensus on the use of unification-based grammar formalisms. Unification-based grammars are basically context-free grammars augmented with additional constraint checking in the form of unification of feature structures. These grammars allow the structure of language to be specified in a declarative way. The research in this thesis largely focuses on applications that are based on this formalism.

Figure 1.1 shows an example of a possible architecture for a speech-based natural language interface. The speech recognizer transforms utterances of the user to text. These words are passed to the parser. The parser looks up all possible interpretations of the words in a dictionary, after which it uses the grammar to derive all possible readings of the sentence. The readings of the sentence are passed for further analysis to the semantic analyzer. At the core of the system, the dialogue manager determines how to respond to the query by generating a response meaning. Finally, the generator transforms this meaning to a sentence, which is then fed to the speech synthesizer. There are many possible variations on this architecture. For example, sometimes the semantical analysis is integrated in the syntactic analysis. The architectures of text-based natural language interfaces, or even machine translation applications, are typically very similar.

1.2 Focus of Research

Parsing of natural language is often one of the most, if not the most, computationally expensive component of natural language processors. Over the past few years, considerable advances in improving the performance of parsers have been made (Oepen and Callmeier, 2000). Nevertheless, the parsing of large sentences can still be computationally demanding. Despite the recent improvements in parser performance, the parsing of a sentence can in some cases still take up well over ten seconds. In applications that require direct user interaction, for example, such processing times are intolerable. In addition, the past has shown that increased performance of parsers has stimulated the development of more complex grammars. The demand for more performance can therefore be expected to remain for some time to come.

Most natural language applications are designed for single processor systems. The increasing availability of multiprocessor systems therefore seems to provide the means to improve the performance of such applications by several factors. For applications that require batch processing of a large number of unrelated sentences or texts, multiprocessor capabilities can trivially be exploited by distributing the sentences of texts amongst a number of independent copies of the parser, running on different processors. However, for other applications, such as applications that require a specific order of processing the sentences, or applications that require direct user interaction, this kind of parallelism is often not a solution. Especially with applications that require direct user interactions, where the processing of a single sentence can be a bottleneck, the capabilities of multiprocessors can only be exploited by parallelizing the internal processes of the respective application.

A considerable amount of research has focussed on parallel natural language processing. An extensive survey of this research was given in (Adriaens and Hahn, 1994). A straightforward approach to parallel processing of natural language is to distribute the different components of an application across processors. Examples of such an approach are the Pangloss system (Nirenburg, 1995) and VERBMOBIL (Amtrup, 1997). With Pangloss multiple equivalent parsing components can be executed in parallel. Obviously, the possibilities for parallelism are limited in this case. Some alternative approaches to natural language processing seem to be more naturally fit for parallel processing. An example of such an approach is memory-based or example-based parsing. With this kind of parsing, a sentence is matched against an existing, often large, corpus of example sentences (Stanfill and Waltz, 1986). These matches can be processed in parallel without ado. Another approach to parsing that seems well-suited for parallelism is actor parsing (Hahn, 1994, Neuhaus and Hahn, 1996). With this approach to parsing, the parser and grammar are defined in terms of actors that communicate through message parsing. An actor can be, for example, one of the lexical entries of a typically highly lexicalized grammar. The often large number of actors can all be run as concurrent processes.

Most of today's large-scale applications for natural language processing, though, use an approach based on unification-based grammar parsing, or similar approach. Adopting another approach to parsing is mostly not an option. In such a case, parallelizing the unification-based grammar parser component is the most obvious approach to improve performance.

Past research on parallel unification-based grammar parsing has yielded moderate results.

One of the difficulties in parallelizing parsers for unification-based grammars is that parsing yields highly unstructured computations. As a result, there can be, for example, large discrepancies in the execution times of tasks. Also, trivial distributions of work amongst processors can easily lead to excess communication to the extent that it thwarts efficient utilization of the computing power. Some researchers have suggested that these problems are insurmountable and that it cannot be expected that effective parallel parsing can become a reality. Recent advances in parallel parsing, however, give more hope in this respect (Ninomiya et al., 2001). As of today, though, there is no consensus on a single best approach to parallel parsing. In this thesis, we will explore the possibilities for a generic and effective solution to parallel parsing of unification-based grammars.

1.3 Context

Since the choice of grammar can greatly influence the performance of a natural language parser, it is important to base the research on a grammar that resembles the current state of the art. Initially, the research was centered around the Deltra grammar, which was developed at the Delft University of Technology. Deltra is a broad-coverage grammar for the Dutch language, and comprises approximately 360 syntactic rules, 120 morphological rules, and a dictionary of about 3000 entries. The grammar is specified in a DCG-like formalism, similar to the DCG apparatus in Prolog. Deltra is a practical grammar that covers a significant part of the Dutch language. Although DCG is used in many existing systems (cf. (Alshawi, 1992), (Briscoe et al., 1987), and (Dowding et al., 1994)), it differs considerably in concept from the formalisms that are typically the focus of today's research. This makes it hard to compare the performance of a parser for Deltra with parsers based on the latter formalisms.

To broaden the scope of the research we later adopted the LinGO grammar in addition to Deltra. The English **LinGO** grammar is a multi-purpose broad-coverage grammar developed at the Center for the Study of Language and information (CSLI) Stanford. The grammar is based on the Head-driven Phrase Structure Grammar (HPSG) framework, a lexicalist approach to linguistic theory that has become one of the dominant theories. The English LinGO grammar specifies approximately 8000 types, 27 lexical rules, 37 syntactic rules, and about 6000 lexical entries.

An advantage of the LinGO grammar is that it has been adopted by multiple research groups, including CSLI Stanford, University of Saarbrücken, and the HPSG group at Tokyo University. Recently two research groups have adopted the LinGO formalism for the development of, respectively, a Japanese and German grammar. Because these grammars are specified in the same formalism, a parser capable of using the English LinGO grammar should require little or no change to use these grammars. This will give even more possibilities for performance analysis and comparison down the road.

To evaluate the performance of a parser, usually a fixed set of sentences is used. Such a collection of sentences is often called a test suite. To compare the performance of different platforms, it is most convenient when the results are based on the same test suite. We have selected three different test sets that we used for benchmarking with LinGO grammar: (i) the often used 'CSLI' test suite with fairly short sentences (Flickinger et al., 1987), (ii) the

suite	# sentences	avg. sen. len.
csli	1348	5.8
aged	96	8.4
fuse	2363	11.6

Table 1.1 Characteristics of test suits used for benchmarking parsers for LinGO.

'aged' test suite, a small collection of sentences of medium length extracted from dialogue utterances derived from a VerbMobil corpus, (iii) the 'fuse' test suite, a balanced extract from four appointment scheduling (spoken) dialogue corpora. The latter suite contains 100 sentences for each input length below 20 words and a limited number of sentences for larger input lengths. Table 1.1 shows the number of sentences and average sentence length for each of the respective test suites. Since LinGO is a more modern and well-known grammar than Deltra, most of the research presented in this thesis focuses on the LinGO grammar. In addition, the availability of standardized test suites and reference platforms for LinGO allows for a more thorough performance analysis.

The empirical results that are presented in this thesis are based on the implementation of two parsers for, respectively, the Deltra and LinGO grammar. Both parsers were specifically written for the research presented in this thesis. The parser for the Deltra grammar was basically a reimplementation of another existing parser for Deltra. The reimplementation was necessary, because the architecture of the old parser proved to be unsuitable for our research goals. Our implementation of the parser was unimaginatively named **Deltra**.

The parser for the LinGO grammar was based on the parser of the PET platform: CHEAP (Callmeier, 2000), and was baptized **CaLi**.[1] PET itself is a reimplementation in C++ of the parser included in the **LKB** system, which is written in lisp (Copestake, 1999). CaLi includes the same optimizations used by CHEAP and therefore yields comparable performance. Basically, CaLi only implements the parser component of PET. The LinGO grammar is provided as a binary file ready to be used for parsing. This means that the algorithms for reading, transforming, and normalizing the grammar did not have to be reimplemented.

1.4 Organization of the Thesis

The next two chapters provide the reader the necessary background for reading this thesis. **Chapter 2** gives an introduction to the field of natural language processing. This chapter is intended both as an introduction to natural language processing for those unacquainted with the field and as a specification of the tools and techniques used for the purpose of our research. **Chapter 3** gives an introduction to the field of parallel processing and scheduling, focusing on the issues that are relevant for our research. The topics addressed include different models of parallel computation, scheduling, optimizing communication, and cache optimizations.

[1] CaLi stands for A *Cha*rt parser for *Lin*GO.

Chapter 4 and 5 both aim at finding the fundamental limitations for parallel parsing by analyzing the extent of the parallelism inherent of parsing and bounds on the amount of communication, respectively. The possibilities and limitations that are derived in these chapters are used as guidelines for the design and implementation of the parallel parser that is presented in Chapter 7.

The remainder of the chapters focus on the implementation of a parallel parser. **Chapter 6** presents a thread-safe unification algorithm. Unification is a common operation in natural language parsers. The most commonly used unification algorithms are not well-suited for parallel processing. The presented algorithm circumvents the problem, without making compromises on performance.

Chapter 7 presents the design and implementation of a parallel parser. The design is largely based on the findings of the presented in the preceding chapters. The chapter concludes with speedup results for runs on a 64 processor shared-memory machine.

Chapter 8 presents several techniques for optimizing the cache performance of parallel and sequential parsers. Improving cache performance reduces traffic on the memory bus. This can be crucial in obtaining the desired performance for multiprocessor implementations.

Finally, **Chapter 9** presents a final discussion of our research and suggestions for future research.

Chapter 2

Natural Language Parsing

Unification-based frameworks have emerged as the dominant formalisms for linguistic theory. With these frameworks, each parsing step includes a constraint check in the form of unification of graphs. Although quite different, Deltra and LinGO are both unification-based grammars. In this chapter, we will explain the concepts and basic algorithms relevant for the sequential and parallel parsing of unification-based grammars.

2.1 Parsing Schemata

Unification-based grammars can be seen as context-free grammars augmented with constraints in the form of feature structures. Although in most modern grammars there is no explicit context-free 'backbone', each parsing step can still be seen as a context-free operation with an additional constraint in the form of unification. In this section, we will present context-free parsing without unification. The addition of feature structures will be discussed in Section 2.2.

The purpose of a parser is to produce parse trees that specify the syntactic structure of a given input string. The way in which a parser constructs its results is determined by both the grammar and the parsing algorithm. A grammar specifies the domain of possible parse trees. The parsing algorithm specifies how such trees can be computed.

There is a great variety of both grammars and parsing algorithms. This variety may make it hard to compare specific characteristics of different parsers. Sikkel (1993b) introduced parsing schemata as a convenient way to specify parsing methods at a high level of abstraction. Parsing schemata allow parsing methods to be specified independent of a particular grammar or particular control, data, or communication structures, while still capturing the essence of the underlying parsing method. The concise way in which parsing schemata allow the essence of parsing methods to be specified has allowed parsing schemata to be used for the *cross-fertilization* of parsing methods. With this cross-fertilization, desirable characteristics of multiple parsing schemata are combined to form an improved parsing method.

As Sikkel (1993b) showed, parsing schemata can easily be extended to parsing methods that deal with grammars beyond the context-free category. In this section we will focus on the context-free part. The extension to unification-based grammars will be discussed in Section 2.2.

In Section 2.1.1, we will introduce parsing schemata in a more formal setting. In Section 2.1.2, we will present mapping relations between parsing schemata, which are useful for defining a taxonomy of parsing schemata. Finally, in Section 2.1.3 and 2.1.4, we

will respectively present some common parsing schemata and parsing schemata specifically designed for parallel processing. The contents of this section have largely been taken from (Sikkel and Nijholt, 1997); for a full formal description of parsing schemata we refer to (Sikkel, 1993b).

2.1.1 Parsing Schemata

Context-free grammars can be considered to be the foundation of parsing schemata. We assume that the reader is familiar with context-free grammars. Nevertheless, for completeness, we will give a brief definition.

Definition 2.1 A **context-free grammar** $G \in \mathcal{CFG}$ is defined as a 4-tuple $\langle N, \Sigma, P, S \rangle$, where N is the set of **non-terminal** symbols, Σ the set of **terminal** symbols, with $N \cap \Sigma = \emptyset$, P the set of **production rules** of the form $N \rightarrow (N \cup \Sigma)^*$, and $S \in N$ the **start symbol**. $\lhd$

We will often write V for $N \cup \Sigma$. In addition, we will use $A, B, \ldots$ to denote any non-terminal in N and we will use $a, b, \ldots$ to denote any terminal in Σ. An arbitrary string in V^* is denoted by $\alpha, \beta, \ldots$. We denote an arbitrary string in Σ^* of length n by $a_1, \ldots, a_n$. Finally, the empty string is denoted by ϵ.

A production $(A, \alpha) \in P$ is also denoted as $A \rightarrow \alpha$. Productions of the form $A \rightarrow \epsilon$ are called **epsilon rules**. The relation $\Rightarrow$ on $V^* \times V^*$ is defined as follows: $\alpha \Rightarrow \beta$ holds if there are $\alpha_1, \alpha_2, A, \gamma$ such that $\alpha = \alpha_1 A \alpha_2$, $\beta = \alpha_1 \gamma \alpha_2$, and $A \rightarrow \gamma \in P$. The transitive and reflexive closure of $\Rightarrow$ is denoted as $\Rightarrow^*$.

We define an **input string** as a sequence of terminals Σ^*. We say a grammar G **recognizes** an input string $a_1, \ldots, a_n$ if $S \Rightarrow^* a_1, \ldots, a_n$ holds. We use $L(G)$ to denote the the set of input strings, or **language**, that is recognized by G.

A **recognizer** for a grammar G can determine whether an input string is an element of the language $L(G)$ produced by that grammar. In addition to recognizing a string, a **parser** also constructs a parse tree in the case a string is recognized. Obviously, parsing a string is a more difficult task than merely recognizing a string. Nevertheless, we will often only consider recognizers when referring to parsers, because it is often straightforward to convert a recognizer into a parser or to obtain a parse tree from the data structures that were already built by the recognizer.

An important subclass of context-free grammars is defined by the class of grammars that can be written in **Chomsky Normal Form**, denoted $\mathcal{CNF}$. A grammar G is an element of $\mathcal{CNF}$ iff all $p \in P$ are either of the form $A \rightarrow a$ or $A \rightarrow BC$. Every context-free grammar G for which $\epsilon \notin L(G)$ can be rewritten into an equivalent grammar in $\mathcal{CNF}$.

An example of a context-free grammar is given in Figure 2.1. It contains two syntactic rules. The first rule defines a sentence (*s*) to be composed of a noun phrase (*np*) followed by a verb (*v*). The second rule defines a noun phrase to consist of a determiner (*det*) followed by a noun (*n*). The grammar also specifies four lexical entries, one determiner, one noun, and two different inflections of the same verb. Note that the grammar is in Chomsky normal form. Obviously, this grammar is nowhere near being a usable grammar for English. It solely

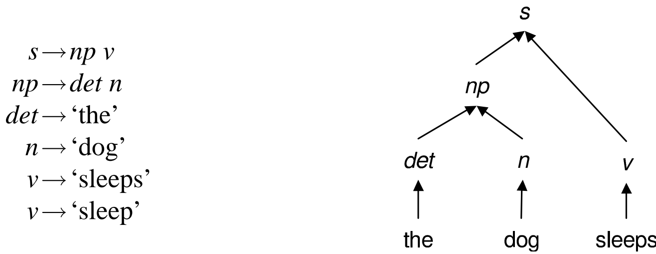


Figure 2.1 Example of context-free grammar and parse tree for the input string "the dog sleeps".

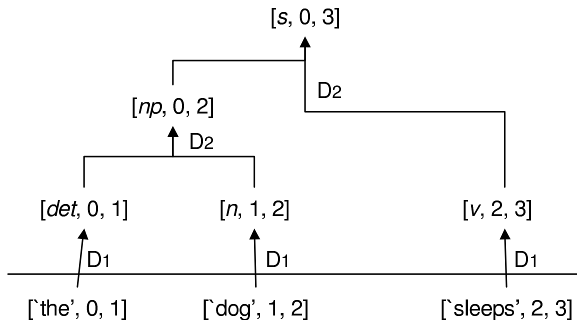


Figure 2.2 Derivation tree for the sentence "the dog sleeps" for **CYK**.

inteded for illustrational purposes. Figure 2.1 also contains a possible parse tree for the sentence "the dog sleeps". Usually, the parsing of a sentence yields more than one complete parse tree, or **reading**.[1] In this simplified example, there is only one possible parse tree.

Parsing schemata are inspired by the "item-based" approach to parsing. According to this point of view, recognizing is a process of deducing a final set of items from an initial set of items by means of a set of deduction rules. Different parsing methods use different item domains, deduction rules, and sets of initial items. Most parsing methods, including Earley-, LR-, and unification-style parsing methods, can effectively be interpreted as item-based parsers.

The application of a parsing schema to a specific grammar and input string is called a parsing system. Figure 2.2 exemplifies a parsing system for the CYK parsing method (Kasami, 1965, Younger, 1967), applied to the grammar given in Figure 2.1. CYK is defined for grammars in Chomsky normal form, The set of initial items, or hypotheses, given an input string $a_1, \ldots, a_n$, is constructed as follows

$$\mathrm{H} \;=\; \{[a, i-1, i] \;\, | \, a = a_i \wedge 1 \leq i \leq n\}.$$

[1]An often quoted example of an ambiguous sentence with multiple readings is "John saw the girl with the telescope".

In the example, the hypotheses are put below the horizontal line. The set of items recognized by CYK, or item domain of CYK is denoted $\mathcal{I}_{\text{CYK}}$. The items in the domain are of the format $[A, i, j]$, where $A \in N$ and $0 \leq i \leq j \leq n$. It holds that $\mathcal{I}_{\text{CYK}} = \{[A, i, j] \mid A \Rightarrow^* a_{i+1}, \ldots, a_j\}$. The input string is recognized if $[S, 0, n]$ is in $\mathcal{I}_{\text{CYK}}$.

The items in $\mathcal{I}_{\text{CYK}}$ are derived by recursively applying inference rules to the hypothesis and the derived items until a fixed point is reached. The derivation process is specified as a set of deduction steps. This approach allows certain restrictions on the inference rules to be defined by simply defining restrictions on this set. For CYK, the set of deduction steps is specified as follows

$$\begin{aligned}
\mathrm{D}^{(1)} &= \{[a, i-1, i] \vdash [A, i-1, i] \mid A \rightarrow a \in P\} \\
\mathrm{D}^{(2)} &= \{[B, i, j], [C, j, k] \vdash [A, i, k] \mid A \rightarrow BC \in P\} \\
\mathrm{D}_{\text{CYK}} &= \mathrm{D}^{(1)} \cup \mathrm{D}^{(2)}
\end{aligned}$$

In Figure 2.2 it is shown how these steps lead to the derivation of the item $[s, 0, 3]$, which represents a complete parse tree of the sentence “the dog sleeps”.

The triple $\langle \mathcal{I}_{\text{CYK}}, \mathrm{H}, \mathrm{D}_{\text{CYK}} \rangle$ defines the parsing system $\mathbb{P}_{\text{CYK}}$ for G and $a_1, \ldots, a_n$. The parsing schema **CYK** for CYK is defined as a generalization of $\mathbb{P}_{\text{CYK}}$ for arbitrary strings and grammars in $\mathcal{CNF}$. In general, we can define parsing systems as follows.

Definition 2.2 A **parsing system** $\mathbb{P}$ for some grammar G and input-string $a_1 \ldots a_n$ is a triple $\mathbb{P} = \langle \mathcal{I}, \mathrm{H}, \mathrm{D} \rangle$, in which

- $\mathcal{I}$ is the **item domain** or item set of $\mathbb{P}$, which specifies the allowed items. (The details of the syntax of items may be different per schema.)
- H is a finite set of initial items, or **hypotheses**.
- $\mathrm{D} \subseteq \wp_{fin}(\mathrm{H} \cup \mathcal{I}) \times \mathcal{I}$ is a set of **deduction rules**, where $\wp_{fin}(\mathrm{H} \cup \mathcal{I})$ represents all finite elements in the power-set of $(\mathrm{H} \cup \mathcal{I})$.

$\triangleleft$

Note that H need not be a subset of $\mathcal{I}$.

An **uninstantiated parsing system** for a grammar G is a function that assigns a parsing system to any $a_1 \ldots a_n \in \Sigma^*$. A uninstantiated parsing system is defined by a triple $\langle \mathcal{I}, \mathcal{H}, \mathrm{D} \rangle$, where $\mathcal{H}$ is a function that assigns a set of hypotheses to a string $a_1 \ldots a_n \in \Sigma^*$. The function $\mathcal{H}$ is usually defined as $\mathcal{H}(a_1 \ldots a_n) = \{[a, i-1, i] \mid a = a_i \wedge 1 \leq i \leq n\}$. A **parsing schema** for some (sub)class of context-free grammars $\mathcal{CG} \subseteq \mathcal{CFG}$ is a function that assigns an uninstantiated parsing system to any grammar $G \in \mathcal{CG}$. We will denote a parsing schema corresponding to a parsing algorithm P as **P**.

In (Sikkel and Nijholt, 1997), parsing schemata are always specified by means of a parsing system, after which this parsing system is generalized for all grammars and input strings. In the remainder of this section, we will adopt this approach and implicitly assume that the definition of parsing systems hold for all $G \in \mathcal{CG}$, for some class of grammars $\mathcal{CG}$, and all input strings $a_1 \ldots a_n \in \Sigma^*$. Next we will define the notions of an inference relation and deduction sequence.

Definition 2.3 For a given $\mathbb{P} = \langle \mathcal{I}, \mathrm{H}, \mathrm{D} \rangle$, the **inference relation** $\vdash \subseteq \wp_{fin}(\mathrm{H} \cup \mathcal{I}) \times \mathcal{I}$ is defined by

$$Y \vdash \xi \text{ if } (Y', \xi) \in \mathrm{D} \text{ for some } Y' \subseteq Y. \tag{2.1}$$

A **deduction sequence** for a parsing system $\mathbb{P} = \langle \mathcal{I}, \mathrm{H}, \mathrm{D} \rangle$ is a pair $(Y; \xi_1, \ldots, \xi_{i-1}) \in \wp_{fin}(\mathrm{H} \cup \mathcal{I}) \times \mathcal{I}^+$, such that $Y \cup \{\xi_1 \ldots \xi_{i-1}\} \vdash \xi_i$, for $1 \leq i \leq j$. $\lhd$

We will use $Y \vdash \xi_1 \vdash \ldots \vdash \xi_j$ as a convenient notation for a deduction sequence $(Y; \xi_1, \ldots, \xi_j)$. The transitive and reflexive inference relation $\vdash^* \subseteq \wp_{fin}(\mathrm{H} \cup \mathcal{I}) \times \mathcal{I}$ is defined as $Y \vdash^* \xi$ if $\xi \in Y$ or $Y \vdash \ldots \vdash \xi$.

2.1.2 Mappings between Parsing Schemata

Once it is known how one parsing method transforms into another, it is sometimes possible to apply similar transformations to other methods. This allows desirable characteristics of one method to be transferred to another.

One type of mapping between parsing schemata is filtering. The goal of a filter is to reduce the number of steps that is needed to complete the parsing. This can be accomplished by both reducing the number of items and reducing the number of deduction steps. It is often possible to discard items or deduction steps without weakening the generality of the parser. Such optimization can lead to more efficient algorithms, but often also to more complicated descriptions of the parsing schema.

Sikkel (1997) distinguishes three different classes of filtering: static filtering, dynamic filtering, and step contraction. With static filtering redundant items or deduction steps of the schema are simply discarded. Dynamic filtering allows taking context information into account. For example, if we know that a partial parse, represented by an item ξ, is only correct if ζ is a valid item, we can replace the reduction steps $\eta_1, \ldots, \eta_k \vdash \xi$ by $\eta_1, \ldots, \eta_k, \zeta \vdash \xi$. An example of dynamic filtering is *look-ahead*. Finally, the most powerful of the filtering methods is step contraction. With step contraction, sequences of reduction steps are replaced by single reduction steps. It is the most powerful of the filtering methods. The filters are defined as follows

Definition 2.4 We define the following filtering relations on any two parsing systems $\mathbb{P}_1$ and $\mathbb{P}_2$.

- The **static filtering** relation $\mathbb{P}_1 \xrightarrow{\mathbf{sf}} \mathbb{P}_2$ holds if $\mathcal{I}_1 \supseteq \mathcal{I}_2$ and $\mathrm{D}_1 \supseteq \mathrm{D}_2$.
- The **dynamic filtering** relation $\mathbb{P}_1 \xrightarrow{\mathbf{df}} \mathbb{P}_2$ holds if $\mathcal{I}_1 \supseteq \mathcal{I}_2$ and $\vdash_1 \supseteq \vdash_2$.
- The **step contraction** relation $\mathbb{P}_1 \xrightarrow{\mathbf{sc}} \mathbb{P}_2$ holds if $\mathcal{I}_1 \supseteq \mathcal{I}_2$ and $\vdash_1^* \supseteq \vdash_2^*$.

Given two parsing schemata $\mathbf{P}_1$ and $\mathbf{P}_2$ for a class of grammars $\mathcal{CG}$, then for any filter $\mathbf{f}$, the relation $\mathbf{P}_1 \stackrel{\mathbf{f}}{\Longrightarrow} \mathbf{P}_2$ holds if for all $G \in \mathcal{CG}$, and $a_1 \ldots a_n$, $\mathbf{P}_1(G)(a_1 \ldots a_n) \xrightarrow{\mathbf{f}} \mathbf{P}_2(G)(a_1 \ldots a_n)$. $\lhd$

Note that $\stackrel{\mathbf{sf}}{\Longrightarrow} \subseteq \stackrel{\mathbf{df}}{\Longrightarrow} \subseteq \stackrel{\mathbf{sc}}{\Longrightarrow}$ holds. In the following discussion, we will present several examples of mappings between parsing schemata.

2.1.3 Common Parsing Schemata

In this section, we will discuss some of the most common parsing methods. We will focus on the parsing methods that have been used for our research in particular. In Section 2.1.1, we introduced **CYK** as an example of a parsing schema. The disadvantage of this schema is that it can only be used for grammars in Chomsky normal form. The Earley parsing schema does not suffer from this problem and is one of the best known parsing schemata for general context-free grammars. The Earley parsing method is inherently uni-directional. Its items include the entire production rule. In addition, a dot is inserted in the right-hand side of the production to indicate the part of the right-hand side that has been recognized so far. A dot indicates that the part on the left of the dot has been recognized, whereas the part on the right of the dot still needs to be recognized. As more of the production is recognized, the dot moves to the right.

An item of the form $[A \rightarrow \alpha \bullet, i, j]$ is called a **passive item** or **passive edge**. Because the entire right hand side is recognized, the left hand side indicates a complete constituent of the type A, where $A \Rightarrow^* a_i, \ldots, a_j$. An item of the form $[A \rightarrow \alpha \bullet X\beta, i, j]$ is called an **active item**, or **active edge**, meaning that the corresponding constituent on the left-hand side is still in the process of being recognized. A simple bottom-up variant of Earley is given by the following schema.

Parsing Schema 2.5 (buE)

$$\mathcal{I}_{\text{buE}} = \{[A \rightarrow \alpha \bullet \beta, i, j] \mid A \rightarrow \alpha\beta \in P \wedge 0 \leq i \leq j\}$$

$$\text{H}_{\text{buE}} = \{[a, i-1, i] \mid a = a_i \wedge 1 \leq i \leq n\}$$

$$\text{D}^{\text{Init}} = \{\vdash [A \rightarrow \bullet \gamma, i, i]\}$$

$$\text{D}^{\text{Scan}} = \{[A \rightarrow \alpha \bullet a\beta, i, j], [a, j, j+1] \vdash [A \rightarrow \alpha a \bullet \beta, i, j+1]\}$$

$$\text{D}^{\text{Complete}} = \{[A \rightarrow \alpha \bullet B\beta, i, j], [B \rightarrow \gamma \bullet, j, k] \vdash [A \rightarrow \alpha B \bullet \beta, i, k]\}$$

$$\text{D}_{\text{buE}} = \text{D}^{\text{Init}} \cup \text{D}^{\text{Scan}} \cup \text{D}^{\text{Complete}}$$

It is called **bottom-up** because parsing starts with constructing the leaves of the parse trees, working its way up to the start symbol. In general, there is no need to initialize an item $[A \rightarrow \bullet \gamma, i, i]$ for each production rule.

The original Earley algorithm limits the number of recognized items by only initializing items of the form $[A \rightarrow \bullet \gamma, i, i]$ if there is a direct need for it, that is, if there is an active item $[B \rightarrow \alpha \bullet A\beta, i, j]$. This technique is called **top-down prediction**. The original Earley algorithm can be defined as follows.

Parsing Schema 2.6 (E)

$$\text{D}^{\text{Init}} = \{\vdash [S \rightarrow \bullet \gamma, 0, 0]\}$$

$$\text{D}^{\text{Predict}} = \{[A \rightarrow \alpha \bullet B\beta, i, j] \vdash [B \rightarrow \bullet \gamma, j, j]\}$$

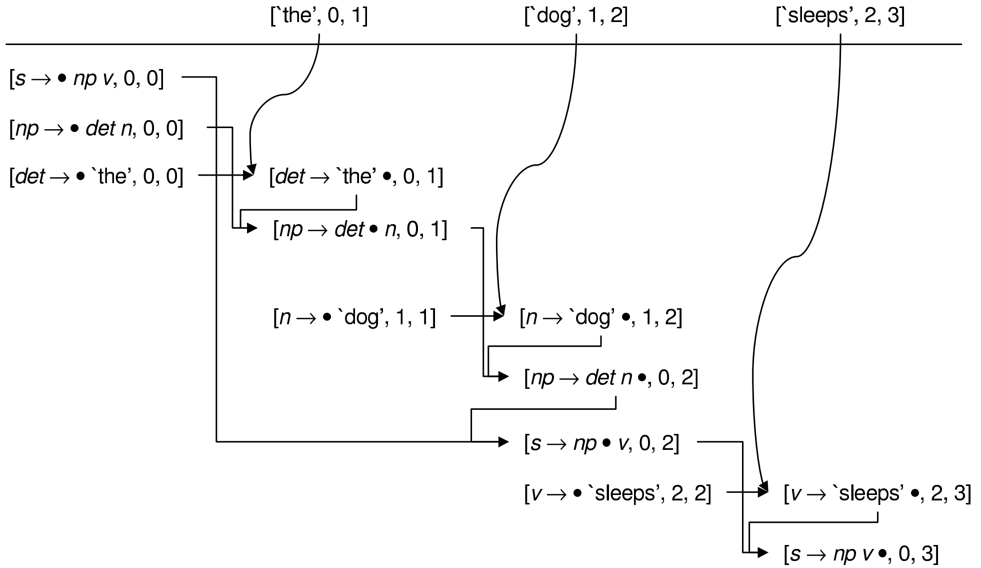


Figure 2.3 Derivation tree for the sentence "the dog sleeps" for **E**. The top left item is derived from $\mathrm{D}^{\mathrm{Init}}$. All items with incoming arcs are derived by either $\mathrm{D}^{\mathrm{Scan}}$ or $\mathrm{D}^{\mathrm{Complete}}$. All other items are the result of $\mathrm{D}^{\mathrm{Predict}}$ and are placed below the item that was used to derive them.

$$\begin{aligned}
\mathrm{D}^{\mathrm{Scan}} &= \{[A\rightarrow\alpha\bullet a\beta, i, j], [a, j, j+1] \vdash [A\rightarrow\alpha a\bullet\beta, i, j+1]\} \\
\mathrm{D}^{\mathrm{Complete}} &= \{[A\rightarrow\alpha\bullet B\beta, i, j], [B\rightarrow\gamma\bullet, j, k] \vdash [A\rightarrow\alpha B\bullet\beta, i, k]\} \\
\mathrm{D}_{\mathrm{E}} &= \mathrm{D}^{\mathrm{Init}} \cup \mathrm{D}^{\mathrm{Predict}} \cup \mathrm{D}^{\mathrm{Scan}} \cup \mathrm{D}^{\mathrm{Complete}}
\end{aligned}$$

Figure 2.3 shows an example of an Earley parsing system, based on the example given in Figure 2.1. As can be seen, items are derived from left to right, with respect to the input sentence.

The $\mathrm{D}^{\mathrm{Init}}$ of **buE** produces more items than the $\mathrm{D}^{\mathrm{init}}$ and $\mathrm{D}^{\mathrm{Predict}}$ of **E**. As a result, **E** yields the following set of recognized items:

$$\{[A\rightarrow\alpha\bullet, i, j] \mid \alpha \Rightarrow^* a_i, \ldots, a_j \wedge S \Rightarrow^* a_1, \ldots, a_i A\gamma \text{ for some } \gamma\},$$

whereas the set of items recognized by **buE** is

$$\{[A\rightarrow\alpha\bullet, i, j] \mid \alpha \Rightarrow^* a_i, \ldots, a_j\}.$$

E can be obtained from **buE** by applying a static filter, i.e., **buE** $\stackrel{\mathbf{sf}}{\Longrightarrow}$ **E**.

Using prediction has the effect of allowing parsers to complete in near linear time, provided that the grammar is "well-behaved". Grammars for natural languages are often not "well-behaved" in this sense, which limits the use of top-down parsing schemata. In addition, the

additional items that are generated by bottom-up parsing can provide useful information in case the sentence is not grammatical. This is useful in case robust parsing is a requirement.

A popular optimization of Earley parsing is Left-Corner parsing. We will describe the underlying parsing method for the generalized left-corner parsing algorithm (c.f. (Matsumoto et al., 1983),(Nederhof, 1993)), as opposed to deterministic left-corner parsing (see (Rosenkrantz and Lewis, 1970)). Generalized LC parsing is a step contraction of Earley: $\mathbf{E} \stackrel{\mathbf{sc}}{\Longrightarrow} \mathbf{LC}$ (Sikkel and op den Akker, 1996). Consider an item $[A \rightarrow B \bullet \beta, i, j]$ in $\mathbb{P}_{\text{buE}}$. This item can only be valid if we already had the valid items $[B \rightarrow \gamma \bullet, i, j]$ and $[A \rightarrow \bullet B\beta, i, i]$, of which the latter is always valid by D_{Init}. So we can replace the steps

$$\begin{aligned} &\vdash [A \rightarrow \bullet B\beta, i, i] \\ [A \rightarrow \bullet B\beta, i, i], [B \rightarrow \gamma \bullet, i, j] &\vdash [A \rightarrow B \bullet \beta, i, j] \end{aligned}$$

by

$$[B \rightarrow \gamma \bullet, i, j] \vdash [A \rightarrow B \bullet \beta, i, j].$$

A similar step contraction can be applied to the scan step. As a result the item domain shrinks considerably. We can transform **E** to a left-corner schema **LC**, using the same transformation. We will consider **buLC** instead of the more elaborate **LC**. **buLC** is defined as follows.

Parsing Schema 2.7 (buLC)

$$\begin{aligned}
\mathcal{I}^{(1)} &= \{[A \rightarrow X\alpha \bullet \beta, i, j] \mid A \rightarrow X\alpha\beta \in P \wedge 0 \le i \le j\} \\
\mathcal{I}^{(2)} &= \{[A \rightarrow \bullet, j, j \mid A \rightarrow \epsilon \in P \wedge j \ge 0\} \\
\mathcal{I}_{\text{buLC}} &= \mathcal{I}^{(1)} \cup \mathcal{I}^{(2)} \\
\text{D}^{\epsilon} &= \{\vdash [A \rightarrow \bullet, j, j\} \\
\text{D}^{\text{LC}(a)} &= \{[a, j-1, j] \vdash [B \rightarrow a \bullet \beta, j-1, j]\} \\
\text{D}^{\text{LC}(A)} &= \{[A \rightarrow \alpha \bullet, i, j] \vdash [B \rightarrow A \bullet \beta, i, j]\} \\
\text{D}^{\text{Scan}} &= \{[A \rightarrow \alpha \bullet a\beta, i, j], [a, j, j+1] \vdash [A \rightarrow \alpha a \bullet \beta, i, j+1]\} \\
\text{D}^{\text{Complete}} &= \{[A \rightarrow \alpha \bullet B\beta, i, j], [B \rightarrow \gamma \bullet, j, k] \vdash [A \rightarrow \alpha B \bullet \beta, i, k]\} \\
\text{D}_{\text{buLC}} &= \text{D}^{\epsilon} \cup \text{D}^{\text{LC}(a)} \cup \text{D}^{\text{LC}(A)} \cup \text{D}^{\text{Scan}} \cup \text{D}^{\text{Complete}}
\end{aligned}$$

Note that the number of steps that is performed by the complete and scan deductions is reduced, because the item domain is limited. The item set can only contain items with a non-empty string left of the dot (except for epsilon rules).

The Deltra parser is based on the so called double dotted parsing schema. Its main characteristic is that it allows parsing not only from left to right, but also from right to left. The double dotted Parser was originally introduced by De Vreught and Honig (1989, 1991). The corresponding parsing schema, **DD**, is defined as follows.

Parsing Schema 2.8 (DD)

$$\begin{aligned}
\mathcal{I}_{\text{DD}} &= \{[A \rightarrow \alpha \bullet \beta \bullet \gamma, i, j] \mid A \rightarrow \alpha\beta\gamma \in P \wedge 0 \le i \le j \wedge (\beta \ne \epsilon \text{ or } \alpha\gamma = \epsilon\} \\
\text{D}^{\text{Init}} &= \{[a, j-1, j] \vdash [A \rightarrow \alpha \bullet a \bullet \gamma, j-1, j]\} \\
\text{D}^{\epsilon} &= \{\vdash [B \rightarrow \bullet\bullet, j, j]\} \\
\text{D}^{\text{Include}} &= \{[B \rightarrow \bullet\beta\bullet, i, j] \vdash [A \rightarrow \alpha \bullet B \bullet \gamma, i, j]\} \\
\text{D}^{\text{Concatenate}} &= \{[A \rightarrow \alpha \bullet \beta_1 \bullet \beta_2 \gamma, i, j], [A \rightarrow \alpha\beta_1 \bullet \beta_2 \bullet \gamma, j, k] \vdash [A \rightarrow \alpha \bullet \beta_1 \beta_2 \bullet \gamma, i, k]\} \\
\text{D}_{\text{DD}} &= \text{D}^{\text{Init}} \cup \text{D}^{\epsilon} \cup \text{D}^{\text{Include}} \cup \text{D}^{\text{Concatenate}}
\end{aligned}$$

A valid item $[A \rightarrow \alpha \bullet \beta \bullet \gamma, i, j]$ represents the fact that $\beta \Rightarrow^* a_{i+1}, \ldots, a_j$ has been recognized. An algorithm implementing this parsing method will perform many redundant operations. In the worst case, a valid item of the form $[A \rightarrow \alpha \bullet \beta \bullet \gamma, i, j]$, with $|\beta| = m$, can be $m-1$ times the result of a concatenation. Similarly, each of the items for the respective productions may have been obtained in various ways as well. Sikkel (1993b) presents several optimizations of **DD** that reduce the number of generated items. An enhancement of this algorithm with look-back and look-ahead filters can be found in (de Vreught and Honig, 1989).

CaLi, the parser for LinGO, uses a **key-driven** parsing schema (Oepen and Carroll, 2000b). Basically, each production rule has one designated non-terminal on its right-hand side that is marked as the key. Instead of recognizing a production from left to right, recognition starts at the key. Since LinGO is a unification-based grammar, a match at the context-free level does not automatically imply a valid item.[2] The idea is to reduce the number of items by choosing the non-terminal as the key that is most-likely to yield a failed unification. The key non-terminal is marked by underlining it in the production rule. The resulting parsing schema is defined as follows.

Parsing Schema 2.9 (KD)

$$\begin{aligned}
\mathcal{I}_{\text{KD}} &= \{[A \rightarrow \alpha \bullet \beta_1 \underline{B} \beta_2 \bullet \gamma, i, j] \mid A \rightarrow \alpha\beta_1 \underline{B} \beta_2 \gamma \in P \wedge 0 \le i \le j\} \\
\text{D}^{\text{Init}} &= \{[a, j-1, j] \vdash [A \rightarrow \bullet \underline{a} \bullet, j-1, j]\} \\
\text{D}^{\text{Include}} &= \{[B \rightarrow \bullet\beta\bullet, i, j] \vdash [A \rightarrow \alpha \bullet \underline{B} \bullet \gamma, i, j]\} \\
\text{D}^{\text{Append}} &= \{[A \rightarrow \beta_1 \bullet \beta_2 \bullet B \beta_3, i, j], [B \rightarrow \bullet\alpha\bullet, j, k] \vdash [A \rightarrow \beta_1 \bullet \beta_2 B \bullet \beta_3, i, k]\} \\
\text{D}^{\text{Prepend}} &= \{[B \rightarrow \bullet\alpha\bullet, i, j], [A \rightarrow \beta_1 B \bullet \beta_2 \bullet, j, k] \vdash [A \rightarrow \beta_1 \bullet B \beta_2 \bullet, i, k]\} \\
\text{D}_{\text{KD}} &= \text{D}^{\text{Init}} \cup \text{D}^{\text{Include}} \cup \text{D}^{\text{Append}} \cup \text{D}^{\text{Prepend}}
\end{aligned}$$

After the key has been instantiated, extending the item at the left dot is only allowed if the right dot is at the end of the production. Allowing extension in only one direction prevents

[2] Unification-based grammars are discussed in Section 2.2.

the generation of duplicate items. Obviously, this order of completing the active item is arbitrary. However, as the right-hand sides of the rules of the English LinGO grammar contain at most two non-terminals, there is only one way to complete an active item. For this reason, the choice of such a scheme has no effect when used with the version of LinGO we used. Because the formalism of LinGO does not allow epsilon rules, we had to include the corresponding deduction rules. Note also that we assumed that terminals do not occur in right-hand sides along with non-terminals, which holds for the English LinGO grammar used for our research.

2.1.4 Parsing Schemata for Parallel Parsing

One of the factors that influences the time complexity of many parsers is the depth of the parse tree. The depth of a parse tree can vary from $\log_2 n$ to n. Rytter's parser has the interesting property that it can parse in $O(\log_2 n)$ time using $O(n^6)$ processors, independent of the depth of the parse tree (Rytter, 1986).

Rytter's parsing method is a step refinement[3] of **CYK**. The step refinement ensures the existence of a balanced parse tree. This is accomplished by introducing many redundancies. This explains the large number of processors that is required. Since Rytter is a step refinement of **CYK**, it is only defined for $\mathcal{CNF}$ grammars. It is possible, however, to apply a similar step refinement to any item-based parser (Sikkel, 1993b). A new parallel algorithm for the resulting parsing methods can parse in polylogarithmic time, provided that a sufficient number of processors is available.

Rytter refines **CYK** by replacing $\mathrm{D}^{(2)}_{\mathbf{CYK}}$ with

$$[B, i, j] \vdash [A, i, k; C, j, k]$$
$$[A, i, k; C, j, k], [C, j, k] \vdash [A, i, k]$$

Items of the form $[A, i, j]$ represent the recognition of a part of the input string $a_i, \dots, a_j$ for a non-terminal A. Items of the form $[A, h, k; B, i, j]$ represent the recognition of a part of the input string $a_h, \dots, a_k$ for a non-terminal A, except for the part $a_i, \dots, a_j$, which still needs to be filled with a substring recognized by $B \Rightarrow^* a_i, \dots, a_j$. These partial specifications of recognized parse trees allow drawing early conclusions of facts yet to be established. This prevents the depth of the parse trees to be of influence on the time complexity.

Rytter's schema is defined as follows.

Parsing Schema 2.10 (Rytter)

$$\mathcal{I}^{\text{recognized}} = \{[A, i, j] \mid A \in N \wedge 0 \leq i < j\}$$
$$\mathcal{I}^{\text{conditional}} = \{[A, h, k; B, i, j] \mid [A, h, k] \in \mathcal{I}^{\text{recognized}} \wedge [B, i, j] \in \mathcal{I}^{\text{recognized}} \wedge h \leq i < j \leq k \wedge (h \neq i \vee j \neq k)\}$$

[3] Step refinement is the inverse of step contraction (Sikkel, 1993b)

$$
\begin{aligned}
\mathcal{I}_{\text{Rytter}} &= \mathcal{I}^{\text{recognized}} \cup \mathcal{I}^{\text{conditional}} \\
\mathrm{D}^{\text{Activate1}} &= \{[B,i,j] \vdash [A,i,k;C,j,k] \mid A \rightarrow BC \in P\} \\
\mathrm{D}^{\text{Activate2}} &= \{[C,j,k] \vdash [A,i,k;B,i,j] \mid A \rightarrow BC \in P\} \\
\mathrm{D}^{\text{Combine}} &= \{[A,h,m;B,i,l],[B,i,l;C,j,k] \vdash [A,h,m;C,j,k]\} \\
\mathrm{D}^{\text{Pebble}} &= \{[A,h,k;B,i,j],[B,i,j] \vdash [A,h,k]\} \\
\mathrm{D}_{\text{Rytter}} &= \mathrm{D}^{(1)}_{\textbf{CYK}} \cup \mathrm{D}^{\text{Activate1}} \cup \mathrm{D}^{\text{Activate2}} \cup \mathrm{D}^{\text{Combine}} \cup \mathrm{D}^{\text{Pebble}}
\end{aligned}
$$

There is one interesting parsing method in between **CYK** and **Rytter** that is particularly well-suited for parallel on-line parsing. With on-line parsing, the input string is presented to the parser token by token, typically in a left to right order. This gives rise to the idea of restricting Rytter items to allow only open parts on the right. This radically reduces the number of possible items. The resulting parsing method, **OCYK** (Sikkel, 1993a), is given in the following schema.

Parsing Schema 2.11 (OCYK)

$$
\begin{aligned}
\mathcal{I}^{\text{recognized}} &= \{[A,i,j] \mid A \in N \wedge 0 \leq i < j\} \\
\mathcal{I}^{\text{conditional}} &= \{[A,h,k;B] \mid A,B \in N \wedge 0 \leq i < j\} \\
\mathcal{I}_{\text{OCYK}} &= \mathcal{I}^{\text{recognized}} \cup \mathcal{I}^{\text{conditional}} \\
\mathrm{D}^{\text{Propose}} &= \{[B,i,j] \vdash [A,i,j;C] \mid A \rightarrow AB \in P\} \\
\mathrm{D}^{\text{Combine}} &= \{[A,i,j;B],[B,j,k;C] \vdash [A,i,k;C]\} \\
\mathrm{D}^{\text{Recognize}} &= \{[A,i,j;B],[B,j,k] \vdash [A,i,k]\} \\
\mathrm{D}_{\text{OCYK}} &= \mathrm{D}^{(1)}_{\textbf{CYK}} \cup \mathrm{D}^{\text{Propose}} \cup \mathrm{D}^{\text{Combine}} \cup \mathrm{D}^{\text{Recognize}}
\end{aligned}
$$

The relation **Rytter** $\overset{\textbf{sc}}{\Longrightarrow}$ **OCYK** $\overset{\textbf{sc}}{\Longrightarrow}$ **CYK** holds. A parser based on this parsing schema can parse in $O(1)$ per word of the input string, using $O(n^2)$ processors.

2.2 Unification-Based Grammars

In the previous section, we discussed approaches to the parsing of purely context-free languages. The dominant linguistic theories for natural language processing applications of today, however, are based on unification-based grammar formalisms. Both LinGO and Deltra are unification-based grammars. There are many small differences between the unification formalisms on which the respective grammars are based, though. Since Deltra's formalism can easily be expressed in terms of the formalism used by LinGO, we will focus the discussion on LinGO and explain the differences as appropriate.

Figure 2.4 Example of a graph unification matrix or attribute matrix.

Unification-based grammar parsing can be seen as context free parsing where each deduction step is augmented with an additional constraint check. Each item and each grammar rule is augmented with a typed feature structure. The constraint check consists of unifying the feature structures of the items and grammar rules involved in a deduction step. If a unification succeeds, the result graph is associated with the newly derived item. Examples of these grammars are **Lexical-Functional Grammar (LFG)** (Bresnan and Kaplan, 1982) and **Head-Driven Phrase Structure Grammar (HPSG)** (Pollard and Sag, 1994).

2.2.1 Unification of Typed Feature Structures

The discussion in this section is based on the introduction of typed feature structures given by Copestake et al. (1999). Later in this section we will explain how this approach differs from the unification mechanism used by Deltra.

Definition 2.12 A **typed feature structure** is a directed acyclic graph with labeled arcs and typed nodes that can be described by a tuple $F = \langle Q, q_0, \delta, \tau \rangle$ where,

- Q is a finite set of nodes,
- $q_0 \in Q$ is the root node,
- $\delta : \mathsf{Feat} \times Q \rightarrow Q$ is the partial feature value function, or the **transition function**,
- $\tau : Q \rightarrow \mathsf{Type}$ is the total node typing function,

where Feat is a finite set of feature names and Type is a set of types. $\lhd$

Figure 2.4 shows two different representations of the same typed feature structure. One is a graph representation, the other is a so called attribute value matrix. An **attribute value matrix** is a notation for feature structures that is often used for HPSG based grammar formalisms. Each pair of brackets represents a **complex node**: a node with at least one child node. Each pair of brackets contains the type of the node and its features. Each feature is followed by the value of the child node. If the child node is complex, it is followed by a set of brackets, otherwise by the type definition. The numbered squares indicate that all features with an equally numbered square share the same value.

A **path** in the graph can be defined as a sequence of features. The domain of possible paths is defined as $\mathsf{Path} = \mathsf{Feat}^*$. The empty path is denoted as ϵ. An extended transition

function $\delta : \mathsf{Path} \times Q \rightarrow Q$ can be defined as follows: $\delta(\epsilon, q) = q$, and $\delta(f \cdot \pi, q) = \delta(\pi, \delta(f, q))$. In the example, the path F.D from the root leads to the node with type $\mathbf{e}$. All nodes should be reachable from the root node. Expressed in terms of the transition function, for any typed feature structure $F = \langle Q, q_0, \delta, \tau \rangle$ it should hold that for all $q \in Q$ there exists a path $\pi \in \mathsf{Path}$ such that $q = \delta(\pi, q_0)$. When two different paths lead to the same node, we call this a **reentrancy**. A feature structure that contains one or more reentrancies is called **reentrant**.

We also define an alternative representation of a feature structure.

Definition 2.13 For any typed feature structure $F = \langle Q, q_0, \delta, \tau \rangle$, its **abstract typed feature structure** is defined as a tuple $\langle \equiv_F, \mathcal{P}_F \rangle$, where

- $\equiv_F \subseteq \mathsf{Path} \times \mathsf{Path}$ where $\pi \equiv_F \pi'$ if and only if $\delta(\pi, q_0) = \delta(\pi', q_0)$ (**path equivalence**) and
- $\mathcal{P}_F(\pi) = t$ if and only if $\tau(\delta(\pi, q_0)) = t$ (**path value**).

$\triangleleft$

Note that it is not strictly necessary to rule out cycles. Some formalisms allow cycles. Ruling out cycles, though, often allows for simpler definitions and more efficient implementations. The formalisms of the LKB and Deltra systems specifically disallow cycles to be specified in the grammar. With LinGO, cycles can still be formed during parsing. The resulting feature structures are never created, however, as the occurance of a cycle is defined to imply a unification failure.

Before we define the unification and subsumption relations on typed feature structures, we need to elaborate on the type system. The advantages of incorporating a type system includes having the concept of inheritance and classification, and enabling error-checking. The details of the type system are not relevant for the discussion in this thesis. We limit our discussion to what is needed to understand the unification operation itself. The type system of the LKB is similar to that presented by Carpenter (1992).

Definition 2.14 The **type system** is a tuple $\langle \langle \mathsf{Type}, \sqsubseteq \rangle, C \rangle$, where $\langle \mathsf{Type}, \sqsubseteq \rangle$ is a type hierarchy, and C a constraint function. The **type hierarchy** $\langle \mathsf{Type}, \sqsubseteq \rangle$ defines a partial order $\sqsubseteq$ on the types in Type. Type includes an element $\top$ for which holds that for all types $t \in \mathsf{Type}, t \sqsubseteq \top$. The **constraint function** C associates feature structure with every type. $\triangleleft$

The type hierarchy $\langle \mathsf{Type}, \sqsubseteq \rangle$ is used to determine the consistency of types. Types are consistent if they share a common subtype. More precisely, a set of types $S \subseteq \mathsf{Type}$ is defined to be **consistent** if and only if there is a type $t_0 \in \mathsf{Type}$ for which it is the case that $t_0 \sqsubseteq t$ for all $t \in S$. The LKB type system specifies two types that are incomparable in the hierarchy to be inconsistent, unless a common subtype is specified. This is often refered to as the closed world assumption. As a result, whenever two types, say $\mathbf{a}$ and $\mathbf{b}$, are consistent, there exists a greatest lower bound, denoted $\mathbf{a} \sqcap \mathbf{b}$. Inconsistency between types $\mathbf{a}$ and $\mathbf{b}$ is denoted $\bot$.

The constraint function C associates a **constraint** feature structure with every type. This feature structure specifies both which features are appropriate for such a node and which features a node should have. The **appropriate features** for a type t is defined as the set of all features leaving the root note of the constraint $C(t)$. More precisely, given a feature structure $F = \langle Q, q_0, \delta, \tau \rangle$ with root q_0, the appropriate features are the features $\text{FEAT}(C(t), q_0)$, where $\text{FEAT}(F, q)$ is defined as a set of features where $f \in \text{FEAT}(F, q)$ if $\delta(f, q_0)$ is defined. The constraints implicitly define when a feature structure is well-formed. A typed feature structure $F = \langle Q, q_0, \delta, \tau \rangle$ is considered to be **well-formed** if and only if for all $q \in Q$, it holds that $F' = \langle Q', q, \delta, \tau \rangle \sqsubseteq C(\tau(q))$ and the appropiate features of $\tau(q)$ equals $\text{FEAT}(F, q)$.

We have now all the necessary knowledge to define the subsumption and unification relations.

Definition 2.15 A typed feature structure F **subsumes** a typed feature structure F', written $F \sqsubseteq F'$, if and only if

- if $\pi \equiv_F \pi'$ then $\pi \equiv_{F'} \pi'$ and
- if $\mathcal{P}_F(\pi) = t$ then $\mathcal{P}_{F'}(\pi)$ where $t' \sqsubseteq t$.

We say that F' is an **instance** of F iff $F' \sqsubseteq F$. The unification relation $\sqcap$ on two typed feature structures can be defined in terms of the subsumption relation as follows. Consider two typed feature structures F and F' and a set $\mathcal{F}$ of feature structures for which holds that $F'' \in \mathcal{F}$ if and only if $F'' \sqsubseteq F$ and $F'' \sqsubseteq F'$. The **unification** of two typed feature structures F and F', denoted $F \sqcap F'$ is defined as the greatest lower bound of $\mathcal{F}$ ordered by subsumption, that is for all $F''' \in \mathcal{F}$ it holds that $F''' \sqsubseteq F''$. If the set $\mathcal{F}$ is empty, unification fails. A failing unification is indicated with $\perp$. $\triangleleft$

An example of a unification of two typed feature structures is given in Figure 2.5. Unification of two feature structures, of respectively type $\mathbf{a}$ and $\mathbf{b}$, can only succeed if $\mathbf{a} \sqcap \mathbf{b}$ exists. If $\mathbf{a} \sqcap \mathbf{b}$ exists, and unification succeeds, the resulting feature structure is of type $\mathbf{a} \sqcap \mathbf{b}$. The unification of two typed feature structures has the result of accumulating all paths of the respective inputs in a single result feature structure. If both feature structures have a common path where one or more of the equivalent nodes on the paths have inconsistent types, than the typed feature structures are considered incompatible and the unification fails.

Unification as defined above does not ensure that the feature structure resulting from unifying two well-formed feature structures is itself well-formed. Suppose we unify two graphs, of respectively type $\mathbf{a}$ and $\mathbf{b}$, with $\mathbf{a} \sqcap \mathbf{b} = \mathbf{c}$ and the appropriate features for $\mathbf{a}, \mathbf{b}$, and $\mathbf{c}$ are $\{F\}, \{G\}$, and $\{F, G, H\}$, respectively. After unifying two well-formed graphs of type $\mathbf{a}$ and $\mathbf{b}$, the resulting graph F_3 will be of type $\mathbf{c}$, but $\text{FEAT}(F_3, q_0)$ will be $\{F, G\}$. Obviously, F_3 is not well-formed as it misses the feature H. To ensure feature structures are well-formed after unification, all nodes q in the resulting graph should be unified with the constraint graph $C(\tau(q))$.

The dialect of unification used by Deltra differs considerably from that of LinGO. Whereas LinGO uses graph unification, Deltra uses term unification. With term unification, each node is associated with a fixed number of arguments, rather than features. Two nodes cannot

$$\begin{bmatrix}\mathbf{a} & \\ \text{F} & \boxed{1}\top \\ \text{G} & \boxed{1}\end{bmatrix} \sqcap \begin{bmatrix}\mathbf{b} & \\ \text{F} & \begin{bmatrix}\mathbf{u} & \\ \text{J} & \mathbf{a} \\ \text{L} & \top\end{bmatrix} \\ \text{G} & \begin{bmatrix}\mathbf{u} & \\ \text{J} & \top \\ \text{K} & \mathbf{b}\end{bmatrix}\end{bmatrix} \Rightarrow \begin{bmatrix}\mathbf{c} & \\ \text{F} & \boxed{2}\begin{bmatrix}\mathbf{u} & \\ \text{J} & \mathbf{a} \\ \text{K} & \mathbf{b} \\ \text{L} & \top\end{bmatrix} \\ \text{G} & \boxed{2}\end{bmatrix}$$

Figure 2.5 A unification with attribute matrices. Note that $\mathbf{a} \sqcap \mathbf{b} = \mathbf{c}$ holds.

unify when the number of arguments does not match. Term unification can be expressed in terms of graph unification simply by defining each type $t \in$ Type to be a child of $\top$. As a result, $\mathbf{a} \sqcap \mathbf{b}$ exists if and only if $\mathbf{a} = \mathbf{b}$. The fact that appropriate features are fixed for each type emulates a fixed number of arguments; the number of arguments cannot grow after unification as there are no subtypes.

2.2.2 Unification-based Grammars

The LinGO formalism expresses almost every aspect of the grammar in terms of feature structures, including the production rules, start symbols, and lexical entries. The type system can be seen as the foundation on which all feature structures are build. For example, all linguistic entities inherit from a single type. The categories corresponding to the non-terminals of a possible context-free grammar (such as $\mathbf{np}$ and $\mathbf{vp}$) have a corresponding type in the type system as well.

All production rules, or **grammar rules**, are encoded as feature structures. The category, or non-terminal, of the left-hand side is encoded in the feature CATEGORY. The elements of the right-hand side of the production are put in the ARGS feature, which leads to a list of arguments. The elements in the list are put in the same order as in the production rule.

Also lexical entries are represented as feature structures, The lexical entries that are associated with the words of the input string are treated by a parser as passive items.

Finally, LinGO allows the definition of multiple start symbols, or **roots**. Because a start symbol specifies a type, each start symbol is associated with a constraint feature structure. An item $[F, 0, n]$ represents the recognition of a sentence of the grammar if it unifies with at least one of the root feature structures.

The DCG formalism used for Deltra differs considerably from that of LinGO. The syntax in which the grammar is described shows more likeness to the common notation for context-free grammars. Figure 2.6 shows a simple example how the same rule can be represented in the different formalisms. Note that the grammar rules in this example are based on the grammar rules in Figure 2.1, augmented with additional constraints. In this specific example, the constraints enforce noun verb agreement.

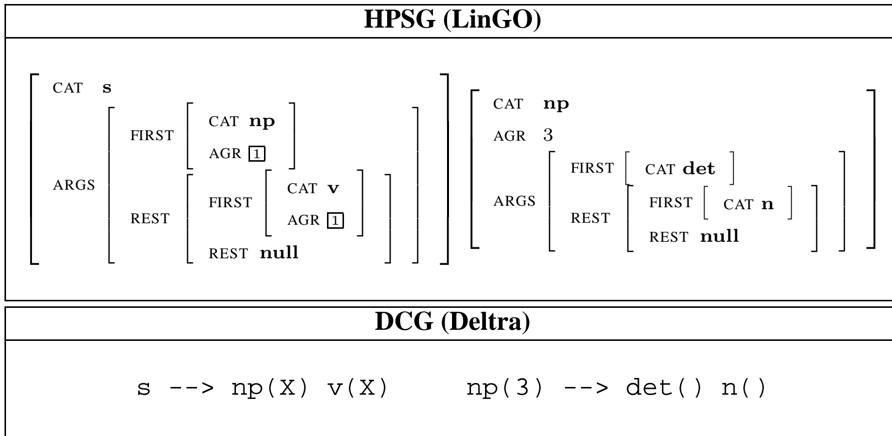


Figure 2.6 Two grammar rules expressed in the HPSG (top) and DCG (bottom) formalisms.

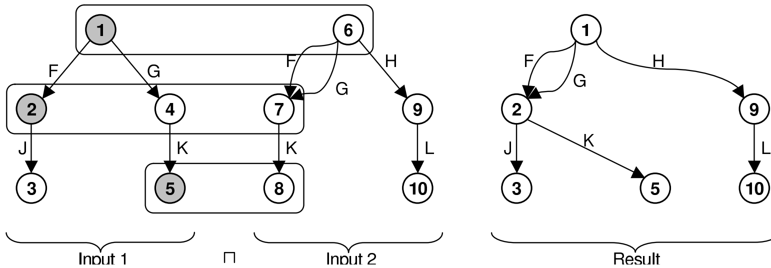


Figure 2.7 Destructive graph unification. Representatives are marked gray.

2.2.3 Algorithms for Unification

A simple algorithm to unify two graphs is the UNION-FIND algorithm (Aho et al., 1974), (Sikkel and Nijholt, 1997). We illustrate this algorithm by means of an example. The example, taken from (Malouf et al., 2000), is shown in Figure 2.7. First, the root nodes are combined into a single equivalence class, labeled $\{1, 6\}$. One node, in this case node 1, is chosen to be the **representative** of the class $\{1, 6\}$. Then all arcs leaving any of the nodes in the class are collected at the representative. If an arc has a unique label within the respective collection of arcs (H in the example), it can simply be added to the representative. If there are two arcs with the same label in the set, then the algorithm recursively descents into the subgraphs pointed to by the respective arcs. In the example, processing the duplicate arcs for F leads to unifying nodes 2 and 7, resulting in the new equivalence class $\{2, 7\}$. Subsequently, unifying the nodes for the arcs labeled G (node 4 and $\{2, 7\}$) results in the equivalence class $\{2, 4, 7\}$. If any incompatibilities are found, unification fails. If the graphs are successfully traversed, the representative of the root equivalence class will have been

UNIFY(dg1, dg2)

1. *Dereference* dg1 *and* dg2.
2. **if** dg1 = dg2 **then**
   return dg1
3. new ← COMPLEMENTARCS(dg1, dg2)
4. shared ← INTERSECTARCS(dg1, dg2)
5. *Set* dg2 *to be the representative of* dg1.
6. **for** *all* (a, b) ∈ shared **do**
   - 6.1. **if** UNIFY(a, b) *succeeds* **then**
     Replace arc b *of* dg2 *with the result.*
   - 6.2. **else**
     return *failure*
7. **for** *all* arc ∈ new **do**
   - 7.1. *Add the arc to* dg2

Figure 2.8 Destructive graph unification

transformed into the root of the result graph.

An example algorithm that implements this approach, taken from (Wroblewski, 1987), is shown in Figure 2.8. Dereferencing a node will yield the representative of the equivalence class a node belongs to or the node itself if the node does not yet belong to any equivalence class. INTERSECTARCS returns the arc labels for features that are defined for both dg1 and dg2. COMPLEMENTARCS(dg1, dg2) gives the set of arc labels that are unique to dg1.

Obviously, UNION-FIND modifies at least one of the input graphs in the process of unification. This unification algorithm is therefore called **destructive**. As unification-based grammar parsers typically retain items and their associated feature structures, additional measures should be taken to prevent the input graphs from getting destroyed. A straightforward solution to this problem is to make a copy of each input graph before unification, leaving the original intact. However, this results in a lot of redundant work. Copying the input graphs means that all nodes of the input graphs need to be visited at least once, even if this was not strictly necessary for unification. This especially holds when a unification is bound to fail. Since failing unifications are common in natural language processing, this solution is unacceptable.

A lot of research has focussed on designing efficient unification algorithms that eliminate the need for excessive copying. One of the earliest attempts was made by Pereira (1985). With his algorithm, the result of unification is a structure called the environment, which specifies how to obtain the result graph in terms of the input graphs. The result graph can be constructed from this data without destroying the input graphs. The disadvantage of this algorithm is that creating the result graph incurs a $O(\log(n))$ overhead for each node visited, where n is the number of nodes. Karttunen (1986) solves the copying problem by storing the destructive changes made to the graphs in an environment. If a unification succeeds, the

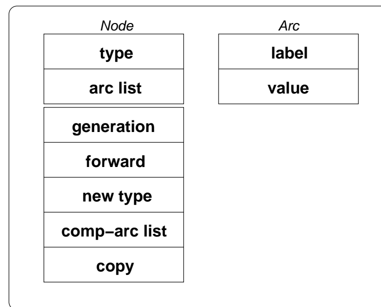


Figure 2.9 Node and arc structure for Tomabechi's algorithm

new graph can be created by copying the graph from the temporarily modified input graphs. After unification completes, the input graphs can be restored from the environment.

Wroblewski (1987) introduced a non-destructive algorithm that eliminates the need to maintain an environment. Basically, the result graph is created on the fly whenever a destructive change to one of the input graphs is about to be made. Each node has an additional copy field to link the nodes in the input graph to the new representative for the duration of the unification. Each node also includes a **generation** field which indicates the validity of the copy field iff it is equal to some global counter. This allows all copy fields to be invalidated after unification simply by incrementing the counter. Wroblewski's algorithm is considered to be very effective. It is often referred to as **incremental copying** or **lazy copying**. Other algorithms that follow this approach are presented in (Kogure, 1990), (Godden, 1990), and (Emele, 1991). The problem with all of these algorithms, though, is that they inevitably create copies unnecessarily in case unification fails.

Tomabechi's quasi-destructive graph unification (1991) eliminates superfluous copying altogether. It is often considered to be one of the most efficient algorithms for unification in natural language applications. It is similar to Wroblewski's algorithm, but instead of creating new nodes during unification, the structure of the new node is included in dedicated fields of the original node, non-destructively. The only copying that takes place is copying of the result graph after unification succeeds. The resulting node and arc structures are shown in Figure 2.9. As with Wroblewski's algorithm, the changes are invalidated after unification by incrementing a global generation counter.

Figure 2.10 shows a version of this algorithm adapted for the typed unification-based grammar of LKB. The forward fields are used during unification to link a node to the representative of its equivalence class. In UNIFY1, dg1 is chosen to be the representative (Step 7). DEREFERENCE follows the forward fields of the nodes, as long as they are valid with respect to the generation. MAKEWELLFORMED ensures that the right set of features is associated with the unified node, given its new type. MAKEWELLFORMED accomplishes this by interleaving a call to UNIFY1 with dg1 and the constraint for the respective new type as its arguments.[4] The function INTERSECTARCS determines the set of arcs with common labels

[4]This was not part of Tomabechi's original algorithm, but was noted by Malouf et al. (2000).

```
UNIFY(dg1, dg2)
  1. if UNIFY1(dg1, dg2) then
       1.1. copy ← COPY(dg1)
       1.2. Increase the generation counter.
       1.3. return copy
  2. else
       2.1. Increase the generation counter.
       2.2. return nil

UNIFY1(dg1, dg2)
  1. dg1 ← DEREFERENCE(dg1)
  2. dg2 ← DEREFERENCE(dg2)
  3. if dg1 ≡_Addr dg2 [a] then
       return true
  4. dg1.newType ← dg1.newType ⊓ dg2.newType
  5. if dg1.newType = ⊥ then
       return false
  6. else
       6.1. if not MAKEWELLFORMED(dg1) then
              return false
       6.2. dg1 ← DEREFERENCE(dg1)
  7. dg2.forward ← dg1
  8. if Any of the nodes has arcs then
       8.1. shared ← INTERSECTARCS(dg1, dg2)
       8.2. for each (r1, r2) in shared do
                if not UNIFY1(r1, r2) then
                  return false
       8.3. new ← COMPLEMENTARCS(dg1, dg2)
       8.4. for each arc in new do
                Push arc to dg1.comp_arcs
  9. return true

DEREFERENCE(dg)
  1. if dg.foward ≠ nil ∧ dg.generation = generation then
       return DEREFERENCE(dg.forward)
  2. else
       return dg
```

[a] Quick equality check based on the nodes' address.

Figure 2.10 The unification algorithm.

COPY(dg)
1. dg $\leftarrow$ DEREFERENCE(dg)
2. **if** dg.copy $\neq$ *nil* **then return** dg.copy
3. newcopy $\leftarrow$ **new** *Node*
4. newcopy.type $\leftarrow$ dg.newType
5. dg.copy $\leftarrow$ newcopy
6. **for each** dg1 **in** dg.arcs $\cup$ dg.comp_arcs **do**
    6.1. dg2 $\leftarrow$ COPY(dg1)
    6.2. *Push* (label, dg2) *into* newcopy.arcs
7. **return** newcopy

Figure 2.11 Copy algorithm

COPY1_{sh}(dg)
1. dg $\leftarrow$ DEREFERENCE(dg)
2. **if** dg.copy $\neq$ *nil* **then return** dg.copy
3. newcopy $\leftarrow$ **new** *Node*
4. newcopy.type $\leftarrow$ dg.newType
5. dg.copy $\leftarrow$ newcopy
6. share $\leftarrow$ dg.type = dg.newType $\wedge$ dg.comp_arcs = $\emptyset$ $\wedge$ ISSAFE(dg)
7. **for each** (label, dg1) **in** dg.arcs $\cup$ dg.comp_arcs **do**
    7.1. dg2 $\leftarrow$ COPY$_{sh}$(dg1)
    7.2. *Push* dg2 *into* newcopy.arcs
    7.3. **if** dg1 $\neq$ dg2 **then**
        share $\leftarrow$ *false*
8. **if** share **then return** dg
    else return newcopy

Figure 2.12 Copy for Tomabechi's algorithm with structure sharing.

for the input nodes. The function COMPLEMENTARCS returns the arcs with labels that exist in dg2, but not in dg1.

When unification succeeds, UNIFY constructs the result graph from the information stored in the input graphs by calling COPY. COPY is shown in figure 2.11. Because all equivalence classes have been determined during unification, the DEREFERENCE in COPY is sufficient to ensure that only one node is created for each node in the result graph. A more elaborate discussion of this algorithm can be found in (Tomabechi, 1995).

Until now we have assumed that a complete copy of the result graph needs to be constructed each time a unification succeeds. Under some circumstances, however, we can allow subgraphs of the input graphs identical to the result to be shared. This sharing of subgraphs is called **structure sharing**. An example of a result graph with structure sharing, again taken from (Malouf et al., 2000), is shown in Figure 2.13. Nodes 3, 5, and 9 are shared in the result. Note that the result graph is equivalent to the result graph in Figure 2.7. Structure

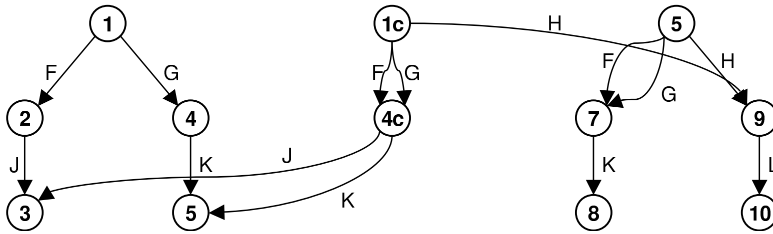


Figure 2.13 Example of structure sharing

sharing can result in a significant reduction of the number of nodes that are copied.

Tomabechi described a structure sharing variant of his algorithm in (Tomabechi, 1992). Malouf et al. (2000) presented an improved version of this algorithm. Using Malouf et al.'s algorithm, a node cannot be shared if any of the following holds:

1. the node has been forwarded to another node
2. the type of the node has changed
3. the node has one or more descendants that need to be copied
4. the node is part of the grammar.

The first three conditions indicate that the result subgraph will not be identical to the original, and hence should be copied. As Malouf et al. (2000) point out, sharing of graphs that are part of the grammar (e.g. lexical entries and rules) can lead to spurious cyclic structures or structure sharing. A structure sharing algorithm based on Tomabechi's approach should implement a way to detect grammar nodes, and copy them accordingly.

The complexity of structure sharing can be hidden entirely in COPY. A version of COPY adapted to allow structure sharing is shown in Figure 2.12. The conditions for structure sharing are checked in Steps 6 and 7.3. ISSAFE returns whether a node is *not* part of the grammar and hence can safely be shared. An implementation of this algorithm should take care to release newcopy to the free pool before exiting when the node can be shared. This can be done efficiently with a mark-release mechanism. Note that because the feature structures are well-formed, there is no need to check for the comp_arcs to be empty. However, checking for the comp_arcs field allows for optimizations that leave feature structures temporarily underspecified.

2.2.4 Parallel Unification

Most unification-based grammar parsers spend about 90% of their time performing unifications (Malouf et al., 2000), (Tomabechi, 1991). It therefore makes sense to focus on the unification operation when parallelizing such parsers. Basically, we distinguish two levels of parallelism. First, it is possible to parallelize a single graph unification. Given that we are

unifying graph F with graph G, we could allow multiple processors to work on the unification of F and G simultaneously. We will call this **parallel unification**. Another approach is to allow multiple graph unifications to run concurrently. Suppose we are unifying graph F and G in addition to unifying graph F and H. By assigning a different processor to each operation we obtain what we will call **concurrent unification**. Parallel unification exploits parallelism inherent of graph unification itself, whereas concurrent unification exploits parallelism at the context-free grammar backbone. In the remainder of this section, we will focus on parallel unification.

From a theoretical perspective, it seems that speedup gains that can be obtained from parallel unification are limited. Yasuura (1984) and Dwork et al. (1984) showed that unification is log-space complete for $\mathbf{P}$. This means that there is no unification algorithm with a time complexity that is polynomial in the logarithm of the input size, using a polynomial number of processors. Such problems are often considered to be "unparallelizable".

By relaxing the prerequisites for a parallelizable algorithm, Vitter and Simons (1986) give a less pessimistic perspective of parallel unification. They introduce a new complexity class $\mathbf{PC}$ that specifies the subclass of $\mathbf{P}$ of problems for which there is an algorithm where a speedup of more than a constant factor, using a polynomial number of processors, is possible. They also specify a subclass of $\mathbf{PC}$, called $\mathbf{PC}^*$, that contains all problems for which the speedup is proportional to the number of processors. Although the authors assert it is unknown whether UNIFY is in $\mathbf{PC}$, they do give a subproblem of unification which is in $\mathbf{PC}^*$. The restriction for the subproblem is that all graphs should have more vertices than edges.

There has also been some research on applying parallel unification for natural language processing. Tomabechi (1991) mentions that his algorithm is well-suited for parallel unification. Parallelism can be introduced in Tomabechi's algorithm at nodes with more than one arc by letting each arc be processed by a different processor. An implementation of a parallel version of Tomabechi's algorithm is given by Fujioka et al. (1990). This paper gives no concrete speedup results. There have been no successful approaches to parallel unification to date.

2.3 Tabular Parsing

In the general case, a trivial implementation of a unification-based grammars parser could yield an exponential running time. This is because such a parser can easily fall into the trap of repeatedly traversing the same parts of the search space. To avoid repeating identical work, a parser needs some kind of memory of the kind of tasks it has already completed. The term that is often used for the collection of parsing techniques that utilize memory for this purpose is **tabular parsing**. Examples of tabular parsing techniques are chart parsing, dynamic programming for parsing, and recursive-descent parsers with memoization. Although the specifics of these techniques can vary considerably, the basic concept is more or less the same. We will focus on the most popular of these techniques: chart parsing. Both Deltra and CaLi are essentially chart parsers.

2.3.1 Chart Parsing

Chart parsing methods were introduced by Kaplan (Kaplan, 1973) and Kay (Kay, 1980). Kay (Kay, 1985) presented chart parsing for unification grammars in specific. A **chart parser** saves the items it produces during parsing on a **chart**. For unification-based grammar parsing, this means that the item is stored along with its feature structure. Recording items in a chart has several advantages: it prevents the same derivations being repeated. it allows multiple parse trees for the same string to be represented in a concise manner, and it allows the analysis of partial results in case the parse fails. It is possible to store only a selection of the items on the chart. For example, Oepen (2000b) suggest a hyper-active chart parser where only passive items are stored on the chart. A chart that contains both active items and passive items is also called an **active chart**. Basically, chart parsing can be applied to any top-down or bottom-up parsing schema.

A chart parser can repeatedly apply the deduction rules of a certain parsing schema until a fixed point is reached. We call this **exhaustive parsing**. Alternatively, a chart parser can stop after finding a single parse tree, or in general, n parse trees. We call this **n-first parsing**. Each application of a deduction rule actually consists of several steps: the context-free **match** itself, the evaluation of the unification constraint, and the **verification** step to check whether the new item is already contained on the chart. The latter involves executing an equality check of the feature structures for each item on the chart that matches the newly derived item based on context-free grounds. Since intelligent hashing can limit the number of equality checks, we will often consider the entire verification phase as a single operation. Note that CaLi does not implement the verification step. The likelihood of duplicate items using the English LinGO grammar is very small. This is because the incorporation of semantics will almost always render two items to be different. Omitting the constraint check therefore eliminates a lot of unnecessary checks, and simplifies the implementation. When the semantics is ignored, duplicate items are rather common, allowing for significant optimizations (Oepen and Carroll, 2000a). For Deltra, the verification step is mandatory, as the **DD** parsing scheme inevitably produces duplicate items.

It is common practice in chart parsing to use a work queue, called the **agenda**, to influence the order of evaluation. When a new item is matched against items on the chart, this can result in many matches. Instead of recursively descending on each alternative, the hypothesized items are put on an agenda. After all matches have been put on the agenda, the parser continues by selecting an entry on the agenda for further processing.

Using an agenda allows fine control over the order of evaluation. Using the agenda as a stack implements a depth-first search strategy, which corresponds to a recursive-descent chart parser. Using the agenda as a FIFO[5] queue implements a breath-first search strategy. An agenda also allows entries on the agenda to be evaluated in order of priority, where the entries have been assigned priorities according to some characteristic. Such schemes are often used in combination with n-first parsing to ensure, or at least increase the likelihood, that the first n parses are actually the n best candidates.

A typical control flow for a chart parser is shown in Figure 2.14. Basically, a chart parser cycles between unify, verification, and match steps, by repeatedly getting and putting tasks

[5]FIFO stands for first in first out.

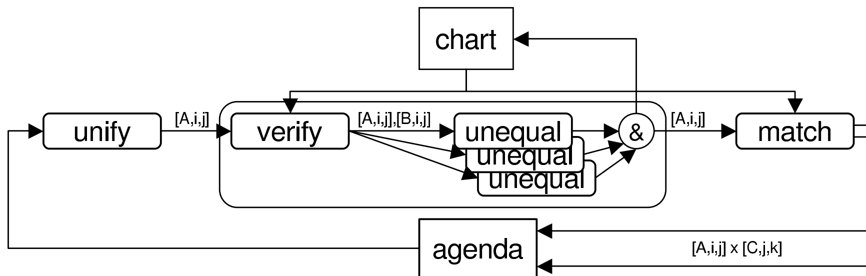


Figure 2.14 Control flow of chart parser.

on the agenda. We will call one iteration of such a cycle a **unify–verify–match cycle**. The parser starts by obtaining a task from the agenda. In the example, the entries on the agenda represent pending unification steps. If the corresponding unification succeeds, the resulting item is passed to the verification step. Here the item is checked for equality with the item on the chart. If the item turns out to be unequal to all of the items, it is stored on the chart and the parser continues with the match step. At the match step, the item is matched against the item based on context-free criteria. For each match, the resulting unification task is put on the agenda. The cycle then starts over. When the cycle is terminated prematurely because of failing unification, a successful equality check, or a failure to match an item with any of the items on the chart, the parser will simply retrieve a new task from the agenda.

An agenda-based chart parser is typically initialized by filling the agenda with work corresponding to the lexical entries. After initialization, a chart parser then simply repeats selecting and processing entries from the agenda until either the agenda is empty or—in case of n-first parsing, sufficient parses have been found.

Note that the agenda can be inserted into different places in the unify–verify–match cycle. With Deltra, for example, the agenda is put just before the match step. This typically reduces the number of entries on the agenda when a breadth-first search strategy is chosen. It does not allow an equally fine control over the order of evaluation though.

2.3.2 Optimizations

If a chart were to be presented as a single list of items, this list would have to be traversed for each match or verification step. Charts are therefore sometimes represented as a upper-triangular matrix, where each cell (i, j) contains items of the form $[A, i, j]$. Such a structure is often called a **tabular chart**.[6] It allows any subset $\{[A, i, b] \mid \forall i \cdot i \leq b\}$, $\{[A, a, j] \mid \forall j \cdot a \leq j\}$, and $\{[A, a, b]\}$ to be traversed without visiting items outside these sets. In general, it is appropriate to split a chart into more specific categories if the lists of items that have to be traversed contain too many items outside the desired category.

Another effective optimization technique is filtering. As mentioned before, in a typical

[6]The CYK algorithm must be the best-known parsing algorithm to use this structure.

unification-based parser over 90% of the unification operations may fail. With **filtering** the goal is to detect failing unifications in an early stage. A simple example of such a filter is the **rule filter** (Kiefer et al., 1999). This filter is based on the observation that if the feature structure for a grammar rule cannot be unified with the argument of another grammar rule, then any instances of the respective rules will not unify either. This allows the construction of a lookup table that quickly identifies whether a unification between any rule and rule-argument combination will fail.

Another important filtering technique is the **quick check** (Malouf et al., 2000). The quick check is based on the observation that certain paths in the feature structures are responsible for a large portion of the failed unifications. The idea is to associate each feature structure with a vector, indicating the values for a set of paths for which is known that they cause the majority of failures. If the path is non-existent in the respective graph, the type is set to $\top$. With the LinGO grammar, the rule filter and quick-check can filter over 90% of the failing unifications. This increases the performance of a parser for LinGO considerably (Malouf et al., 2000). The filters are typically applied before a candidate edge is put on the agenda. This allows the size of the agenda to be reduced considerably.

2.3.3 Parallel Chart Parsing

From Figure 2.14 it is clear that the most important source of parallelism is at the match step. At this step the parsing process may branch of into multiple independent paths of execution. This parallelism can straightforwardly be exploited by using the agenda as a work queue. A critical design decision in such an approach is whether to use a single **centralized agenda** for all processors, or whether to use a **distributed agenda**, where each processor has its own agenda. A distributed agenda can reduce synchronization overhead, because it prevents processor from having to contend for a single structure. However, if the agenda is used as a prioritized queue, a centralized agenda may be a necessity.

A similar design decision is whether to store the items derived by all processors in a single **centralized chart**, or whether to use some kind of **distributed chart**, such that each processor can store the items it derives locally. Many proposals to distribute a chart have been presented. In most proposals, the chart is split on the basis of a classification of the items. Each processor may be assigned one or more of the subcharts. A processor that owns the subchart is typically responsible for executing the work that corresponds to the subchart.

Nijholt (1994) gives an overview of several techniques to split the chart. A well-known approach is to base the distribution on a tabular chart, where a processor $P_{i,j}$ is created for each cell (i, j) in the tabular matrix. $P_{i,j}$ is responsible for applying deduction rules of the form $[A, i, k], [B, k, j] \vdash [C, i, j]$. We call this kind of parser a **parallel tabular chart parser**. Similarly, a **string chart parser** can be obtained by assigning processors $P_{0,i}$ to sets of items $\{[A, 0, i]\}$. A completely different approach is to split the chart based on the grammar rule, or part of the grammar rule, that is represented by the respective items. This approach was taken, for example, by Yonezawa and Ohsawa (1994) for their data-flow like on chip parser design.

When items are not evenly distributed amongst the charts this can lead to an unbalanced work load leading to inefficient use of processor resources. By means of ad hoc load bal-

ancing techniques, though, it is still possible to balance load at the granularity of single tasks Ad hoc load balancing techniques have been proposed by, for example, De Vreught (1992) and Ibarra et al. (1991) (for context-free grammars).

2.4 Existing Parallel Natural Language Parsers

Parallel parsing for NLP has been researched extensively. Nevertheless, many of the presented solutions either did not yield acceptable speedup or were very specific to one application, preventing it from being a practical solution for other parsers. Recent research, however, seems to indicate that an efficient parallel parser is feasible. In this section we will give an overview of past research on parallel parsing.

Initially, research on parallel parsing focussed on bounds on algorithms, rather than implementations. For example, Lozinskii and Nirenburg (1986) presented a parallel tabular parser for locally-sensitive context-free languages with a time complexity of $O(\log^4 n)$ using n processors, where n is the length of the input sentence. De Vreught (1993) presented an overview of fast and slow parsing with a focus on tabular chart parsing. Nijholt (1991, 1994) gives an extensive overview of approaches to parallel chart parsing.

Tomita (1985) presented a parallel LR parser. His LR parser does not have an agenda that can be used to distribute work. Parallelism is introduced whenever ambiguity arises in the input sentence. Tomita's work has mainly been concerned with the creation of an efficient sequential implementation for full context-free parsing. As is argued in (Nijholt, 1991), Tomita's solution is not usable as a solution specifically designed for parallel machines. The graph-structured stack requires a master process. Moreover, each input word needs to be supplied to each process, which is inefficient.

Early research on parallel chart parsing was presented by Grishman and Chitrao (Grishman and Chitrao, 1988). Their algorithm was based on an extended version of the CYK algorithm with top-down prediction for a context-free grammar augmented with procedural constraints. Distribution of work is accomplished through a centralized agenda. The agenda is inserted in the unify–verify–match cycle right before the match phase. This means that one entry on the agenda can correspond to multiple constraint evaluation operations. To avoid a too coarsely grained parallelism, they also provide an implementation that places the finer grained tasks on the agenda. The experiments are based on 4 different grammars of different size. The results are based on runs on the NYU Ultracomputer, an 8 node shared-memory computer, and simulations for runs with up to 40 processors. The authors report that the simulations and real speedups are comparable for up to 4 processors. For more processors, considerably better results are obtained with simulations. Reasonable speedups are obtained for the larger grammars, with a maximum of 15 with 40 processors. The finer-grained parallelism proved to be crucial in obtaining the speedups.

De Vreught et al. (Hoogerbrugge and de Vreught, 1991, Olk and de Vreught, 1992) developed a parallel tabular chart parser for the **DD** parsing schema (see page 15), based on a master slave approach with load balancing. Both describe an implementation for a Meiko with 16 INMOS T800 transputers. They conclude that the number of processors that can effectively be used is limited. Hoogerbrugge and De Vreught (1991) also describe a decen-

tralized approach for the Meiko which exhibited increasing speedup up until the maximum number of possible processors ($n + 1$). Both papers also conclude that tabular chart parsers do not seem to be appropriate for parallelization on the Meiko.

Thompson (1991) presents two implementations of a parallel parser for two different systems. The target architecture for the first implementation was the Connection Machine, a large scale SIMD machine. To exploit the SIMD architecture, the context-free recognition process is mapped on a matrix multiplication problem. The target architecture for the second implementation was the Butterfly, a MIMD processor with a set of MC68000 processors. The parser uses a centralized chart and centralized agenda.

Both access to the agenda and access to the chart is locked in order to avoid race conditions. Prevention of duplicate work is enforced by additional synchronization. Each processor is assigned to handle either a chart or an agenda. Typically one processor is assigned to maintain an agenda and all other processors are assigned to maintain a chart. A maximum speedup of 3.3 is obtained with 4 processors. Thompson also presented other, less successful, implementations of parallel parsers for distributed systems (Thompson, 1991).

Nurkkala et al. (1994a) presented a parallel parser for the UPenn TAG grammar for the KSR1 shared-memory machine, with 64 nodes.[7] The implementation is a parallel version of an Earley chart parser for TAG, with a distributed agenda and centralized chart. To avoid communication overhead, each processor has its own agenda. When processors become idle, they are allowed to take work from other processor's agendas. The authors give an extensive performance analysis based on three grammars: a 256-tree English grammar and two randomly generated grammars of 512 and 1024 trees. Their parser achieves almost near linear speedup for the 1024 tree grammar. For the English grammar, however, speedup does not exceed 10 in terms of wall-clock time or little over 30 with 64 processors in terms of cpu time. The author's note that the results may be somewhat biased, because the processor speed on their experimental platform (20MHz) seemed artificially slow compared to the speed of the interconnect.

Nurkkala et al. (1994b) also presented a TAG parser for a 256 node nCUBE/2. Because the nCUBE/2 does not have shared-memory, they used a different parallelization technique. Basically, the trees of the grammar are assigned to the nodes in a round robin fashion. Each processor maintains its own chart and agenda. The processors maintain a list of derived items, which is periodically broadcasted to other nodes. The results are similar as with the implementation for the KSR1. However, because there is no dynamic load balancing, speedup is somewhat restricted. The maximum speedup obtained with the grammar for English and a 30 word sentence reaches about 12 with 64 or more processors.

Görz et al. (1996) present an implementation of the INTARC parser, a subproject of the Verbmobil research effort. Their parser uses a type check filter that eliminates most unifications. The parser uses a centralized agenda with a priority ordering on the entries, and a centralized chart. The authors mention communication costs as a reason for choosing a centralized chart. The experiments were run on a SparcServer1000 with 6 processors, and consisted of parsing 10 sentences with respectively 2, 3, and 4 processor setups. All 2 and 4 processor runs resulted in slowdown. Their best results indicate a speedup of 1.3 with 3

[7] More precisely, it has a non-uniform memory access (NUMA) architecture.

processors. The authors blame the high optimization level of the sequential parser as one of the reasons for the poor performance. They claim that the type check causes large disparities in the granularity of agenda tasks, which make load-balancing difficult.

Manousopoulou et al. (1997) discuss a parallel parser generator based on the Eu-PAGE system. This system is built on top of Orchid and PVM, using a distributed shared-memory model. This solution exploits coarse-grained parallelism. The authors note that this makes the ability to achieve speedup greatly dependent on the grammar that is being used. As we will see in Section 4, the LinGO and Deltra grammars require a much more fine-grained approach to parallelism (see also Görz et al. (1996)). The authors present speedup results relative to a one processor version for two selected sentences. A maximum speedup of 5.99 on 7 processors is obtained.

Pontelli et al. (1998) show how two existing NLP applications (Ultra and ArtWork) were successfully parallelized using the parallel Prolog environment ACE. ACE combines or-parallelism and independent and-parallelism (Kacsuk, 1990). For the translation application Ultra, speedups of up to 13 were obtained with 15 processors. An advantage of this approach is that also non-parsing components of NLP systems can be parallelized. The disadvantage of this approach, though, is that it can only be applied to parsers developed in Prolog.

Yoshida et al. (Yoshida et al., 1999) presented a 2-phase parallel FB-LTAG parser, where the operations on feature structures are all performed in the second phase. Parallelism is introduced by spawning a new thread for each task that would otherwise be added to an agenda. Spawning of threads is implemented with StackThreads/MP (Taura and Yonezawa, 1999), a parallel programming library which enables the program to create threads with little overhead. The system uses the PSTFS environment (Ninomiya et al., 1998) for the processing of feature structures. The speedup ranged up to 8.8 for 20 processors. Parallelism is mainly thwarted by a lack of parallelism in the first phase.

Ninomiya et al. (2001) developed an agent-based parallel parser based on the parallel tabular chart parser concept of distribution. The cells of the chart are represented by threads and are responsible for doing all the work at the context-free level. The unification operations are performed by separate agents. Each processor is associated with one unification worker. Whenever a unification operation needs to be performed, one of the processors is chosen at random to execute it. The system is implemented in ABCL/f and LiLFeS. The maximum speedup reached was 13.2 with 50 processors.

We would like to note that it is hard to compare the performance of parsers if different grammars are being used. Different grammars can give a very different dynamics to the problem. In addition, the continuous improvement to parsing techniques, such as the quick check, developed over the past decade has changed the dynamics of the problem as well.

Chapter 3

Analysis and Optimization of Parallel Computations

In this chapter, we will give a brief introduction to techniques for the modeling and analysis of parallel computations. The topics addressed include different models of parallel computation, scheduling, optimizing communication, and cache optimizations.

3.1 Parallel Computing

A parallel computer can be seen as a collection of identical, closely interconnected processors to allow the exchange of data and coordination of activities. We distinguish these types of computer from distributed systems, where processors are loosely coupled and possibly spread over a large geographic area. Parallel computers are specifically designed to let their processors jointly solve a single problem.

3.1.1 Parallel Models

We can analyze the possibilities for parallelism for a computation at an abstract level by specifying the computation in a parallel model. There are several ways to model a computation for a parallel computer. A model of parallel computations should have several properties. It should be simple enough to allow an algorithm to be easily specified and analyzed. It should also allow for a graceful transition between design and implementation. Finally, a performance analysis carried out on the model should reflect the performance of an actual implementation of the algorithm. There is no one model that satisfies all these requirements for all applications. In this section, we will present a classification, as presented by JáJá (1992), of the most prominent models.

The Network Model The network model closely relates to multiprocessor architectures where different processors, or nodes, are connected through a network. In this model, each node has its own memory. Communication between processors is established by means of message passing. A processor has to send data to another processor by explicitly sending it a message. The receiving processor has to explicitly call a receive in order to get the message.

The network model is often not well-suited for modeling parallel natural language parsing. With structure sharing, for example, each time two graphs get unified, the result graph may have references to subgraphs of the respective input graphs. Because unification between

graphs stored on different processors is inevitable, such references will extend beyond the local memory. Analyzing the communication that results from such references can be complicated. It can also be expected that an implementation of sharing of subgraphs across a network incurs considerable overhead. A solution in this case could be to allow structure sharing only within one node and to only send complete graphs between processors. Although the network model provides the right level of abstraction in some situations, we will generally prefer a model where we can incorporate constructs like structure sharing without change.

The Shared-Memory Model The shared-memory model is a natural extension of the basic model of sequential computation. Each processor has access to a single shared memory and has no memory of its own. All references to memory are therefore defined to have the same access time. Obviously, as each processor shares the same memory, constructs like structure sharing can be incorporated without change in the parallel model.

The shared-memory model closely relates to the architecture of todays shared-memory multiprocessors. One aspect that the shared-memory model does not model correctly is the fact that the processors of these architectures incorporate caches. One of the effects of using caches is that access to locally generated data often requires less cycles than access to data generated by other processors, because the locally generated may still be cached. A correct model of these machines would therefore be some hybrid of the network model and the shared-memory model.

Task Graphs Computations can be modeled in a completely architecture independent way by means of task graphs. A **task graph** is a directed acyclic graph (dag), where the nodes represent some unit of execution, or tasks, and the arcs represent precedence constraints on the order of execution of the these tasks. Nodes without incoming arcs represent the input of the computation.

An algorithm can typically not be represented by a unique task graph, as the specific tasks that are performed by the algorithm will depend on the input size and, possibly, the specific values of the input. An algorithm can be represented, however, by a set of dags, which contains a representative for each possible input.

Task graphs are particularly suitable for parallel time complexity analysis, as the model clearly specifies which operations can be performed in parallel. The dag model abstracts away from communication costs between processors. Instead it is assumed that each processor can access the memory of any other processor with zero cost. An implementation of an algorithm can be obtained by **scheduling** each node of the graph on a processor at a specific time. More precisely, given P processors, each node i is associated with a pair (p_i, t_i), where $1 \leq p_i \leq P$, indicating that processor p executes node i at time t_i, such that

1. if $t_i = t_j$, for some $i \neq j$, then $p_i \neq p_j$, which means that no processor can execute two nodes simultaneously, and
2. if (i, j) is an arc in the graph, then $t_j > t_i$, ensuring that the precedence constraints be respected.

Input nodes are typically assumed to take 0 time. We call the set $\{(p_i, t_i) \mid i \in N\}$ a **schedule** for the parallel execution of a dag by P processors, where N is the set of nodes in the graph.

The time required to execute a schedule is defined as $\max_{i \in N} t_i$. The **parallel complexity** $T_P(n)$, or T_P, of a dag for P processors is defined as taking the minimum over all schedules for the dags that use P processors.

3.1.2 Performance of Parallel Computations

Obviously, the ultimate goal of solving a problem with a parallel algorithm is to be faster than its sequential counterpart. Let **P** be a problem for which is known that the **sequential complexity** of **P** is $T_{\text{seq}}(n)$, where n is the input size. In other words, there is provably no other sequential algorithm than can solve **P** in less then $T_{\text{seq}}(n)$ time. In addition, let A be a parallel algorithm that can solve **P** in $T_P(n)$ time given P processors. The **speedup** that is achieved by A is then defined as

$$S_P(n) = \frac{T_{\text{seq}}(n)}{T_P(n)}. \tag{3.1}$$

In the ideal case, we want $S_P(n)$ to be P. It is common practice to use the time complexity of the best known sequential algorithm for $T_{\text{seq}}(n)$ in case there is no provably best algorithm.

The **nominal speedup** is defined to be $T_1(n)/T_P(n)$ and gives the speedup relative to the parallel algorithm itself run on a single processor. The nominal speedup is useful to evaluate the efficiency of the utilization of the processors by the algorithm. Dividing the nominal speedup by the number of processors yields the **efficiency** of a parallel algorithm:

$$E_P(n) = \frac{T_1(n)}{PT_P(n)}. \tag{3.2}$$

In this thesis, we will usually use speedup in the sense of nominal speedup, unless indicated otherwise.

In theory, the maximum possible speedup that can be achieved corresponds to the **average available parallelism**, which is defined as $\bar{P} = T_1/T_\infty$ (Frigo et al., 1998). In practice, factors like communication overhead, synchronization overhead, and a lack of parallelism in the problem can thwart this goal.

3.2 Scheduling

To be able to parse an input in parallel, we need to segment it in small executable units. In Section 2.3.3, we noted that the most common approach to parallelize chart parsers is to exploit parallelism at the context-free backbone, yielding concurrent unification. Therefore, we define each deduction step to be a unit of execution, or task. As the derivation of new

items during parsing can initiate new deduction steps, the number of tasks grows dynamically. The purpose of a scheduling algorithm is to assign each of these tasks to one of the available processors. To efficiently execute such a dynamically growing pool of tasks, a scheduling algorithm should ensure that work is evenly distributed amongst all processors. To put bounds on the amount of memory required, a scheduling algorithm should also limit the number of tasks that are spawned simultaneously. Finally, a scheduling algorithm should attempt to schedule related tasks on the same processor so that communication between processors can be reduced.

In this section, we will present the necessary tools to analyze the scheduling of parallel computations. We will also discuss some scheduling algorithms.

3.2.1 Bounds for Optimal Schedulings

In Section 3.1.1, we presented task graphs as a possible model for parallel computations. A task graph allows us to quantify the execution time for a P-processor system. For now, we will assume that all tasks take the same time to complete. We define the **dag depth** of a task to be the length of the longest path terminating at the task. Analogously, we define the **dag depth** of a task graph, denoted T_∞, to be the maximum dag depth of any task. This is also referred to as the **critical path** of a computation. We use T_∞ to denote the dag depth because it indicates the minimum time necessary to complete the computation, given an infinite number of processors. Finally the total **work** of a computation T_1 is defined as the time it takes to complete all tasks of a computation, using one processor. Obviously, because a processor can only execute one task at a time, $T_P \geq T_1/P$ holds.

Brent (1974) and Graham (1969) have derived upper bounds for the execution times of optimal schedules. It follows from their work that for all multithreaded computations, there exists a P-processor execution schedule for which it is the case that

$$T_P \leq T_1/P + T_\infty. \qquad (3.3)$$

In (Blumofe and Leiserson, 1993) these results are extended to show that this upper bound can be achieved with **greedy schedules**. With greedy schedules, at each step of the execution, if at least P tasks are ready, then P tasks execute, and if fewer than P tasks are ready, then all execute. So for any multithreaded computation with work T_1 and dag depth T_∞ and for any number P of processors, any greedy schedule achieves $T_P \leq T_1/P + T_\infty$.

3.2.2 Scheduling Algorithms

A scheduling algorithm specifies how a set of tasks is distributed amongst a set of processors, taking the dependencies between tasks into account. Since the set of tasks required to parse an input is not known beforehand, we need to consider run-time or dynamic scheduling techniques. Obviously, the design of the scheduling algorithm first of all depends on whether the set of tasks are represented as, for example, a stack, a loop definition, or a work queue. For chart parsers, the agenda provides a good point for distributing work.

A lot of research on scheduling has focussed on distributing loop iterations. Scheduling of loop iterations is similar to the scheduling of tasks on a work queue. The major difference is that with loop operations, the number of iterations is usually fixed, whereas with work queues tasks are usually added during processing. Nevertheless, many of the scheduling techniques designed for loop operations also apply to work queues.

One of the simplest scheduling algorithms is **static scheduling**. With static scheduling, the decision about which processor a task should be executed by is based on off-line criteria. Therefore, with this technique, the same tasks will always be assigned the same processor. A parallel tabular chart parser, for example, implements a static scheduling technique. Static scheduling can minimize the run-time synchronization overhead. On the other hand, it can result in a poor distribution of the work load. To avoid this problem, a scheduling algorithm for parallel chart parsing will have to implement some kind of dynamic load balancing, or **dynamic scheduling**.

The major goal of most dynamic scheduling algorithms is to minimize synchronization overhead, while optimizing load balancing. A very simple dynamic scheduling algorithm that achieves near perfect load balancing is called **self scheduling** (Tang and Yew, 1986). With this algorithm, each processor repeatedly executes one task from a centralized queue until the queue is empty. Obviously, all processors will finish within one task from each other. A major disadvantage of this scheme, however, is that the centralized queue needs to be locked each time a task is removed from it. This can result in an unacceptably large synchronization overhead.

The **uniform-sized chunking** algorithm (Kruskal and Weiss, 1985) reduces the synchronization overhead of self-scheduling by letting each processor fetch K tasks at a time. This means that the cost for locking the centralized queue is amortized over the execution of K tasks. Instead of having a fixed chunk size, some scheduling approaches adapt the chunk size to the size of the queue. For example, with **guided self-scheduling** (Polychronopoulos and Kuck, 1987), each processor is assigned $1/P^{\text{th}}$ of the remaining tasks of the queue. This method will typically incur less synchronization overhead than uniform-sized chunking, but will more likely yield an uneven distribution of work.

Markatos and LeBlanc show the importance of cache-awareness for scheduling tasks on different processors. The time a parallel application spends moving data into its local memory or cache can take up 30–60% of the total execution time, yielding a significant source of overhead (Markatos and LeBlanc, 1992b). With **affinity scheduling** (Markatos and LeBlanc, 1992a), the iterations of a loop are divided into $\lceil N/P \rceil$ chunks. Each processor is assigned a fixed chunk. The idea is that a fixed sequence of iterations will typically cause the same data to be accessed on successive visits of the loop. This increases the chance that the memory is already loaded in the cache on each successive visit. To enable load-balancing, work can be moved from one processor to another. The results, however, will be moved back to the original processor, in order to restore affinity. In this way, the overhead of loading data into cache is only incurred when load-balancing occurs.

A completely different approach to scheduling is to distribute the work queue itself. To achieve load balancing, processors can donate work to other processors at given times (**work sharing**) or take work from other processors if they become idle (**work stealing**). In (Blumofe and Leiserson, 1994) it is shown that work stealing schedulers lead to less commu-

nication than work sharing schedulers. Work stealing has also proven to be a successful way to implement a greedy scheduler. Just as with self-scheduling, all processors terminate within one task of each other. The synchronization overhead only occurs when one thread steals work from another thread's queue. As stealing is typically infrequent and contention is localized, the total amount of synchronization overhead is usually reduced considerably.

With work stealing, a thief thread selects a victim thread to steal from according to some **stealing policy**. For example, it can select the thread which has the most work on its queue (Markatos and LeBlanc, 1992a). Such a strategy is typically not well-suited when a lot of processors are used. For large P, random selection of a thief has proven to be useful (Blumofe and Leiserson, 1994), (Dandamudi, 1991).

A work-stealing algorithm for strict multithreaded computations is presented by Blumofe and Leiserson (1994). Their algorithm unifies the approaches of other work-stealing schedulers for, for example, backtrack search computations (Karp and Zhang, 1993) and divide-and-conquer applications (Wu and Kung, 1991). This algorithm has been used in the Cilk scheduler (Frigo et al., 1998). With Cilk, each thread has its own stack and threads are allowed to "steal work" from the *bottom* of the stack of other threads whenever they become idle. The strict multithreaded computations define the class of computations for which the Cilk scheduler can run efficiently. Most importantly, as the authors note, Cilk is not well-suited for applications that require constant synchronization through a shared data structure. Shared data structures are common in chart parsing. In addition, the stack-based approach seems to limit the freedom in order of evaluation, which is often required for parsing. It seems, therefore, that Cilk is not well-suited for parallelizing chart parsers.

3.2.3 Work-first Principle

Frigo et al. (1998) present an important design criterion for their scheduler, called the **work-first principle**. The work-first principle states that overhead borne by the work of a computation should be minimized by moving it on the critical path. This can be explained as follows. First, it is assumed that the scheduler is optimal (i.e. its running time is bound by Inequality 3.3), that there is sufficient parallelism, and that there is a sequential algorithm against which the parallel version can be evaluated. The critical path T_∞ and the average parallelism $\bar{P} = T_1/P$ both present a lower bound on the execution time for T_P. The **critical path overhead** is defined as the smallest constant c_∞ such that

$$T_P \leq T_1/P + c_\infty T_\infty. \tag{3.4}$$

The **work overhead** is defined as $c_1 = T_1/Tseq$, where T_{seq} is the running time of the sequential processor. Under the assumption of sufficient parallelism, we can assume that $\bar{P}/P \gg c_\infty$. From this it follows that $T_1/P \gg c_\infty T_\infty$ and, from Inequality 3.4, we obtain the nominal speedup $T_P \approx T_1/P$. In other words, provided there is enough parallelism, the critical path overhead has little impact on the nominal speedup.

To compare the parallel implementation with the serial implementation, we include the work overhead in Inequality 3.4, giving $T_P \leq c_1 T_{\text{seq}}/P + c_\infty T_\infty$. This simplifies to $T_P \approx c_1 T_{\text{seq}}/P$ under the assumption of sufficient parallelism. We see that minimizing c_1 has a

more direct impact on the speedup. For the work stealing scheduler presented by Frigo et al. (1998), this principle means that the overhead is moved to thieves, rather than workers as much as possible.

3.3 Domain Decomposition

In many applications, communication can be a bottleneck in obtaining sufficient speedup. When there is too much communication between processors, the network, or the memory bus in case of shared-memory systems, can become saturated. This can cause significant overhead for all processors. In this section, we will present some of the available techniques to optimize distribution of work such that communication between processors is minimized.

3.3.1 Introduction

The problem of **domain decomposition**, or **partitioning**, is to divide the tasks of a computation across a given number of processors such that communication is minimized while, simultaneously, keeping the work evenly distributed across the processors.

A common way to model the communication involved in a computation is to use a graph where nodes are labeled with a certain amount of work and edges are labeled with the amount of communication that is required between the nodes. Because the direction of the communication is typically irrelevant for measuring the amount of communication, such graphs are typically represented as undirected graphs,

In the following discussion we assume we are given a graph $\langle V, E\rangle$, a function W that assigns a cost for processing each task in V, a function S that gives a measure for the amount of communication associated with each edge in E, and a partitioning $\mathcal{P}$ that assigns each $v \in V$ to one of P processors. There are two common ways to define communication: the edge cut and the communication volume. The **edge cut** is defined as

$$\sum_{(u,v)\in E} \begin{cases} S((u,v)) & \mathcal{P}(u) \neq \mathcal{P}(v) \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

The edge cut represents a scenario where the processors do not keep a record of messages previously received from other processors. That is, all communication is counted, even when the same data is sent more than once between the same pair of processors.

In scenarios where repeated transmission between two processors can be avoided, the edge cut measure may incorrectly model the communication. In this case, it may be more appropriate to measure the **communication volume**, which is defined as

$$\sum_{v\in V} S(v)\ |\{u \cdot (u,v) \in E\ \ | \mathcal{P}(u) \neq \mathcal{P}(v)\}| \tag{3.6}$$

where S now associates the size of the communicated data with the nodes. Most research has focussed on optimizing the edge cut.

The **balance** of a partitioning $\mathcal{P}$ is typically defined as follows.

$$\frac{\max_{p=1,\ldots,P}\left(\sum_{v\in\{u\in V \mid \mathcal{P}(u)=p\}} W(v)\right)\cdot P}{\sum_{v\in V} W(v)}. \tag{3.7}$$

A perfectly balanced partition will have a balance of 1.

The problem of domain decomposition can now be defined as finding a partitioning $\mathcal{P}$ for a given task graph that simultaneously minimizes the balance and either the edge cut or communication volume. The weights to attach to respectively the measure for the communication and the measure for the balance is arbitrary. One common technique is to define an upper bound for the balance, after which communication is minimized while keeping the balance below this upper bound.

Finding the optimal partitioning can be a computationally expensive operation. Domain decomposition is therefore of most use for application domains where a graph models a repeated pattern of communication within one computation. This is often the case with, for example, finite element analysis. Parsers typically fall outside the class of such applications. However, in Chapter 5, we will use domain decomposition as a tool to analyze the possibilities for minimizing communication in natural language parsers.

3.3.2 Partitioning Algorithms

The presented partitioning problem is known to be NP-complete (Karypis and Kumar, 1998). Given that the number of nodes in graphs can be quite large, it is unreasonable to expect to find optimal partitionings. A lot of research, however, has focussed on finding high quality approximations.

Many approaches to the partitioning problem can only handle 2 processor partitionings (Karypis and Kumar, 1998). An example of a popular bisection algorithms is **Recursive spectral bisection (RSB)** (Barnard and Simon, 1994). The direct computation of a k-way partitioning is typically much harder to compute than a good bisection. For this reason, the bisection algorithms are often used to obtain k-way partitionings by recursively partitioning the obtained partitions. To improve the quality of the resulting partitionings, the partitioning phase is often followed by a refinement phase. The algorithm presented by Kernighan and Lin (1970) is commonly used for this purpose.

As an alternative to recursive bisection, multilevel recursive bisection (MLRB) has been proposed (Hendrickson and Leland, 1995). The idea behind MLRB is to first coarsen a given graph to a graph with a few hundred nodes, compute the partitioning, and then gradually refine the graph to the original graph. Multilevel recursive bisection yields significantly better partitionings than simple recursive bisections at much better efficiency (Karypis and Kumar, 1998). One of the advantages of coarsening is that it becomes feasible to compute a k-way partition directly, rather than computing it through recursive bisection. In (Karypis and Kumar, 1998) a k-way refinement algorithm is presented that can be used in combination with this approach.

3.4 Caching Principles

Improving cache performance of an application can considerably reduce the amount of data a processor has to move to and from memory. Reducing traffic on the memory bus can significantly improve the performance of a single processor system. Reducing traffic, however, can be of even more importance for multiprocessor shared-memory systems. With shared-memory systems, all processors typically share the same memory bus. Obviously, when all processors heavily utilize the bus, the bus can get easily congested. This, in turn, can significantly slow down a computation and nullify the advantage of having multiple processors.

3.4.1 Terminology

Modern-day computers usually have a multiple level memory hierarchy. The memory closest to the processor is called the **level** 1 cache. The next level is called level 2 cache, etc. Typically, lower level memory is faster, but also smaller in size. At higher levels, there usually is more storage capacity at the expense of higher access times. Typical modern-day computers have 2 or 3 levels of cache between processor and main memory.

Most caches map locations in cache to memory locations at the granularity of a **cache line**. The cache line size is typically 2^k times the word size, where $k = 1, 2, \ldots$. To avoid confusion, we will often refer to the cache line sized regions in main memory that map to a single cache line as **memory lines**. When a reference to a location in memory is made, the entire memory line is loaded in the cache, including the unreferenced data. This can be seen as a kind of prefetching.

On shared-memory architectures, caches can be either **shared** or non-shared. Each processor usually has its own level 1 cache. Level 2 caches are sometimes shared between all processors. In case each processor has its own level 2 cache, there may be a shared level 3 cache.

Because a cache at level n typically cannot hold all data of memory at level $n + 1$, there needs to be some administration of which data is actually stored in cache. In addition, there needs to be some policy to determine which memory lines to flush when room should be made for other memory lines. In the ideal case, a cache can map addresses of higher levels to any position in its cache. This allows the full capacity of the cache to be utilized at any point in time. This is called a **fully associative cache**. Usually, however, simpler schemes are used to limit the overhead of finding elements in the cache.

A simple way to eliminate overhead is direct mapping. With **direct mapping**, an element of a higher memory level is always mapped to the same location in cache. Using this scheme, the position in the cache can be derived directly from the element's memory address. Direct mapping can result in very inefficient use of the cache, because several data elements may map to the same location. **Set-associative mapping** is a compromise between direct mapping and fully associative mapping. It still maps a memory address to a fixed position in cache, but allows multiple entries at every position. The number of slots at these positions is fixed. A cache with a slots per position is called an **a-way associative** cache. 2-way and

4-way associative caches are common configurations. When a new entry is entered in the set it needs to be determined which of the old elements is thrown out. Typical policies for this are random selection or the **least recently used (LRU)** element.

The purpose of a cache is to improve memory access times by exploiting the reuse of data in an application. **Reuse** of data occurs when the same data or cache line is accessed more than once in the execution of the program. The reuse is exploited when the data is retained in the cache between successive accesses. We can distinguish two different kinds of reuse: temporal and spatial reuse. **Temporal reuse** occurs when multiple accesses to the same physical address or data element use a buffered copy in the cache or registers. **Spatial reuse** occurs when multiple accesses to data all refer to the same cache line. Obviously, temporal reuse is a subclass of spatial reuse, because a repeated access of the same element implies a repeated access of the same cache line.

A **cache miss** occurs when a data element being accessed cannot be found in the cache. A piece of data may not be stored in the cache for several reasons. The most obvious reason is the case where a memory line is referenced for the first time. We call these types of misses **compulsory misses**. **Interference misses** are the counterpart of compulsory misses, and include all misses that are caused by the fact that reusable cache lines have been flushed from the cache. In other words, the first time a memory line is accessed, it will always cause a compulsory cache miss. Each additional miss that results from accessing the same memory line will be an interference miss.

Interference misses can be further grouped into several categories. **Capacity misses** occur when the cache size is insufficient to store all data that can be reused. For example, suppose one is looping over an array that is twice the cache size. After the entire array has been accessed (in ascending order), the second half of the array will have flushed the first half out of the cache. When the loop is repeated, the first half will flush the second half out of the cache before it can ever be reused. Ultimately, cached data is never reused!

Conflict misses occur when data is flushed from the cache because $a + 1$ memory lines, where a is the cache's associativity, were mapped to the same cache line. In general, the greater the associativity of a cache, the less prone it is to conflict misses.

There are also some effects that can cause interference misses on shared-memory multiprocessors in particular. Many of the shared-memory systems use some kind of coherency protocol to keep the caches of the processors consistent. Whenever a processor writes to a location in main memory, all cached copies of the respective data on other processors need to be invalidated. This will result in a cache miss whenever these processors attempt to reuse the respective data. We refer to these kind of cache misses as **coherency misses**. These additional cache misses caused by coherency enforcement can be seen as the equivalent of communication in distributed systems.

Unnecessary coherency misses can be caused by unintended spatial reuse. Consider two variables, each of which is to be used locally by a different processor. Usually, such variables can simply be cached when a processor needs to access them repeatedly. However, if the two variables happen to be on the same memory line, the coherency protocol will continuously detect conflicts. The variables will need to be reloaded repeatedly. This effect is known as **false sharing**. False sharing can be prevented by ensuring that each memory line contains only non-related data.

$$
\begin{array}{l}
\textbf{for } i = 1 \ldots n \textbf{ do} \\
\quad \textbf{for } j = 1 \ldots mN \textbf{ do} \\
\quad\quad \textit{do something with } A[i,j]
\end{array}
\quad \rightarrow \quad
\begin{array}{l}
\textbf{for } k = 1 \ldots m \textbf{ do} \\
\quad \textbf{for } i = 1 \ldots n \textbf{ do} \\
\quad\quad \textbf{for } j = (k-1)N \ldots kN \textbf{ do} \\
\quad\quad\quad \textit{do something with } A[i,j]
\end{array}
$$

Figure 3.1 Blocking is applied to the left loop to eliminate capacity misses. Here, N is the number of elements of array A that fit in the cache.

3.4.2 Techniques to Improve Cache Utilization

Numerous solutions have been proposed to improve cache utilization. Most of the research focussed on optimizing loop-nests operating on non-sparse arrays and matrices. Nevertheless, in recent years, research interest has increased for more unstructured problems like sparse matrix-vector multiplication and finite element analysis with irregular meshes. Optimizing cache behavior with respect to cache-coherent shared-memory architectures has gained interest as well. In this section, we will highlight the techniques most important for our research.

Blocking As we mentioned, capacity misses occur when the cache size is insufficient to store all data that can be reused. In a worst case scenario, this phenomenon can eliminate all possibilities for temporal reuse. One of the techniques to eliminate capacity misses is **blocking** or **tiling**. Blocking promotes temporal reuse by limiting the active data set to the size of the cache. The technique can best be explained by means of the following simple example. Suppose we have a loop that iterates n times, in ascending order, over each element in an array. The array occupies $k\mathcal{C}$ bytes, where $\mathcal{C}$ is the cache size and $k = 2, 3, \ldots$. As we explained, successive iterations over the complete array will cause all cache lines to be flushed, disallowing any exploitation of temporal reuse. We can eliminate this effect processing the array one block at a time. The transformed loop is shown in figure 3.1. Basically, capacity misses are avoided by processing *all* operations on one block before proceeding to the next.

Obviously, the n elements that were referenced in the original outer loop now have to be reloaded k times. In general, it is often not possible to reduce capacity misses completely. Nevertheless, blocking can still reduce the number of interference misses considerably. Examples of blocking can be found in (Wolf and Lam, 1991).

Locality Grouping Ding and Kennedy (1999) present several techniques to improve temporal reuse for irregularly structured applications. With **locality grouping** temporal reuse is increased by reordering computation. Consider the following example, taken from (Ding and Kennedy, 1999), where a series of operations need to be executed on a machine with a 3-element fully associative cache with LRU replacement. We will assume there is no spatial reuse. An unstructured version of the program executes the following series of operations with two operands:

$$(b\ c)(e\ g)(e\ f)(a\ b)(f\ g)(a\ c)$$

On the given 3-element cache, this sequence will result in 10 misses, of which 4 are interference misses. Now, if we reorder the operations by grouping them, for example, on access to the elements in increasing order, we get:

$$(a\ b)(a\ c)(b\ c)(e\ g)(e\ f)(f\ g)$$

This simple strategy reduces the number of misses to 6 by eliminating all interference misses. In their experiments, the authors show that locality grouping can reduce the number of interference misses by as much as 96%.[1]

The authors also present a similar run-time technique to improve spatial reuse by putting data that is typically accessed simultaneously in a single cache line. In addition, the authors suggest that data should be separated according to when and how it is used. That is, data that is used in separate phases in computation should be stored in different cache lines. In addition, read only and read/write data should be separated to prevent the redundant writing back of read only data.

[1] Their results are based on simulations on a fully associative caching architecture. Conflict misses cannot occur in such an architecture.

Chapter 4

Parallelism in Parsing Computations

In this chapter, we will analyze different approaches to parallel parsing. The aim of the analyses is to identify possible bottlenecks and fundamental limitations of parallel parsing. We will show that to ensure there is sufficient parallelism, parsers should distribute work at the granularity of individual unification steps.

4.1 Introduction

Rather than forming an understanding by means of a complexity analysis of the parsing algorithm and grammar, we will base the analysis on actual parsing. Carroll (1994) showed that the exponential complexities with respect to grammar size and input length often have little impact on the performance of unification-based parsers. He argued that the study and optimization of unification-based parsers should, therefore, rely on empirical data, at least until complexity theory can more accurately predict the practical behavior of parsers. Since we believe that recently no real advances have been made on this front, we follow the approach of basing performance analysis on actual parser behavior. In other words, rather than analyzing parsing schemata or uninstantiated parsing systems, we will base our analyses on parsing systems.

Furthermore, we will investigate parallelism at the context-free level. As we argued in Section 2.2.4, the possibilities for parallel unification are limited by both theoretical and practical limitations. We therefore believe that parallel unification can be used at best in addition to concurrent unification. At least, it is useful to first consider the possibilities of parallelism at the context-free level.

In the next section, we will give an insight into the structure of a parsing computation. In Section 4.3, we will investigate the possibilities for parallelism based on a critical path analysis. This is accomplished by performing analyses on task-graphs constructed from real-life parsing examples. In Section 4.4, we will perform a more abstract analysis to analyze the effect of the choice of parsing schema on the amount of parallelism.

4.2 Parsing as Deduction

Parsing systems are a good starting point to investigate the dynamics of parsing. Basically, all parsers that implement the same parsing schemata will typically yield the same set of active and passive items. The derivation of an item corresponds to a certain task and the

derivation dependencies between items determine the paths of execution. In this section, we will define a graph that captures this information.

4.2.1 Definitions

The justification graph, which we will define next, specifies the dependencies for the derivation of items corresponding to a given parsing system. It is defined in terms of the relation D of a given parsing system.

Definition 4.1 A **justification graph** for an instantiated parsing system $\mathbb{P} = \langle \mathcal{I}, \mathrm{H}, \mathrm{D} \rangle$ (see Definition 2.2) is a hypergraph $\mathcal{G} = (V, D)$, where V is a set of nodes on the domain $\mathrm{H} \cup \mathcal{I}$. V is defined as

$$\{I \in Y \ | (Y, m) \in D\} \ \cup \ \{I \ | (Y, I) \in D\}.$$

The set of deduction rules specifies the arcs of the hypergraph, where for each $(Y, m) \in D$ the source nodes are given by Y and the destination node is given by m. $\lhd$

For example, the justification graphs for **CYK** and **E** corresponding to the example of Figure 2.1, can be visualized by the derivation trees given in Figure 2.2 and 2.3, respectively.

Given a justification graph $\mathcal{G} = (V, D)$. An item $I \in V$ is said to **precede** an item $J \in V$, denoted $I \prec J$, if and only if there exists a relation $(Y, J) \in D$, where $I \in Y$. A **path** from node I to node J is defined as a sequence $\langle n_0, \dots, n_k \rangle$, where $I = n_0$, $J = n_k$, $I, J \in V$, and for each successive pair of nodes n_i, n_{i+1}, it holds that $n_i \prec n_{i+1}$. We will use the notation $I \prec^* J$ to indicate that there exists a path from I to J. Two items $I_a, I_b \in V$ are said to **match**, denoted $I_a \otimes I_b$, if there exists a relation $(Y, I_c) \in D$, where $I_a, I_b \in Y$.

We define three disjunct subsets of V: initialization items, hypotheses, and derived items. Each represents a different phase of the parsing process. The set of **initialization items** V_{Init} of $\mathcal{G}$ is defined as the smallest set such that:

$$V_{\text{Init}} = \{I \ | (Y, I) \in D, \text{ where } Y \subseteq V_{\text{Init}}\}$$

The initialization items represent all items that can be derived independently of the input string. This set will be empty if the grammar does not contain any epsilon rules. The hypotheses are the items corresponding to the input string. The set of **hypotheses** V_{H} of $\mathcal{G}$ is defined as $V_{\text{H}} = H$. The set of **derived items** of $\mathcal{G}$ is simply defined as $V_{\text{D}} = V - (V_{\text{Init}} \cup V_{\text{H}})$.

A justification graph represents which items justify the deduction of another item, recursively. The nodes of a justification graph only specify successfully derived items. That is, it does not include any representation of the work required for attempted matches that failed. The justification graph therefore indicates the work done for a parser with perfect filtering. Since all parsers that use the same parsing schema will typically produce the same set of items, the justification graph provides an appropriate level of abstraction to investigate properties of parsing schemata independent of specific implementations.

4.2.2 Strict Parsing Systems

In this section, we will define a class of parsing systems for which it is straightforward to determine worst-case distributions. We will use these findings to prove the validity of a metric that we will present in the next section. We will prove that parsing systems based on a grammar without epsilon rules and either the parsing schema **DD** or **KD** are members of this class.

Definition 4.2 We will call an instantiated parsing system $\mathbb{P} = \langle \mathcal{I}, \mathrm{H}, \mathrm{D} \rangle$, represented as the justification graph $\mathcal{G} = (V, D)$, **strict** if for all $I, J \in V$ it holds that if $I \prec^* J$, then not $I \otimes J$. $\lhd$

It is often useful to consider a slightly weaker variant where this property only holds for the subset $(V_{\mathrm{D}} \cup V_{\mathrm{H}})$. If for all $I, J \in V_{\mathrm{D}} \cup V_{\mathrm{H}}$ it holds that if $I \prec^* J$ then not $I \otimes J$, then we call $\mathbb{P}$ strict on $V_{\mathrm{D}} \cup V_{\mathrm{H}}$. Note that these two properties are equivalent if the grammar on which the parsing system is based has no epsilon rules. Finally, we call an uninstantiated parsing system strict if, for all inputs, the resulting instantiated parsing system is strict.

Next we will prove that all uninstantiated parsing systems based on **DD** and **KD** and a grammar without epsilon rules are strict.

Theorem 4.3 *Given a grammar G without epsilon rules, then all uninstantiated parsing systems for G based on the double dotted parsing schema* **DD** *are strict.*

Proof. We need to prove that for all possible epsilon rule free grammars and for all input sentences, there are no two nodes I and J in the justification graph $\mathcal{G} = (V, D)$ of the corresponding parsing system, for which $I \prec^* J$ and $I \otimes J$ hold simultaneously. We need to consider which sequence of deduction steps can yield such a condition.

Because the grammar does not contain any epsilon rules, we do not have to consider D^{ϵ}. In addition, this means that each item $a \in V$ covers a non-empty range of the input string. For the remaining deduction rules it holds that the result item spans a part of the input string that at least includes the span of any of the left-hand side items. This means that if $[\alpha, i, j] \prec^* [\beta, k, m]$ then $k \leq i$ and $m \geq j$ will hold.

Now, the only deduction step that can produce a match between two items is $\mathrm{D}^{\mathrm{Concatenate}}$. Consider the following match by concatenation: $[\alpha, i-a, i] \otimes [\alpha, i, i+b]$. Since each item covers a non-empty part of the input string, we know that $a > 0$ and $b > 0$. Obviously, neither $[\alpha, i-a, i] \prec^* [\alpha, i, i+b]$ nor $[\alpha, i, i+b] \prec^* [\alpha, i-a, i]$ can hold, because respectively $i \not\leq i-a$ and $i \not\geq i+b$. $\square$

Note that this proof only considers context-free matches and does not include the unification constraint. In other words, if $I \prec^* J$, I will fail to match J on context-free grounds, without considering the additional unification constraint. Assuming that a parser will never attempt to unify the feature structures of two items if the items do not match on context-free grounds, the feature structures of I and J will never be unified.

Theorem 4.4 *Given a grammar G without epsilon rules, then all uninstantiated parsing systems for G base on the key driven parsing schema* **KD** *are strict.*

Proof. This proof is analogous to the proof for **DD**. We note that we did not make use of the dot positions or the matching of grammar rules between the items in $\mathrm{D}^{\mathrm{Concatenate}}$ in the proof for the double dotted schema. This means that the same arguments that hold for $\mathrm{D}^{\mathrm{Concatenate}}$ will also hold for $\mathrm{D}^{\mathrm{Complete}}$. □

We note that in both cases it is possible to construct a grammar containing epsilon rules that will result in non-strict parsing systems.

4.3 Task-Graph Analysis

Since a justification graph only indicates successful derivations, it gives a simplified picture of what operations are performed during parsing, In reality, each implementation of a parser is bound to do some extra work. For example, failed unifications and equality checking are not included as specific tasks in the justification graph.

In this section, we will give a more refined representation of the work that is carried out by a parser by means of task dependency graph. We will use this representation to derive the average available parallelism that exists when parsing. This information can give us an upper bound for the speedup that can be obtained using dynamic load balancing techniques for shared-memory architectures. As we saw in Section 3.2.1, the average parallelism in a computation can be derived by dividing the total amount of work T_1 by the critical path length T_∞ of its respective task graph.

4.3.1 Task Dependencies

To analyze parsing computations in more detail we use a task dependency graph. A task dependency graph specifies *all* work that is done by a parser, including failing unifications and match operations. Compared to a justification graph, a task dependency graph specifies the computation carried out while parsing in more detail and removes the ambiguities concerning the order of execution (think of cycles, items derived multiple times, etc.).

Definition 4.5 A **task dependency graph** $T = \langle \mathcal{T}, E, W \rangle$ is a directed acyclic graph (dag). Each node in $\mathcal{T}$ represents a task. A **task** is an atomic unit of computation. E is a set of directed edges indicating dependencies between tasks: if $(a, b) \in E$ then a should complete before b may be executed. Finally, W is a function defined as $W : \mathcal{T} \rightarrow \mathbb{N}$ that assigns a cost to each task $T \in \mathcal{T}$. ◁

Justification graphs specify a parsing computation at a more abstract level than task dependency graphs. The details of a task dependency graph greatly depend on the specifics of the implementation of the parser. For this reason, there can be many task dependency graphs that correspond to the same justification graph. A task dependency graph T is said to correspond to a justification graph $G = \langle V, D \rangle$ if it complies with the following requirements:

1. $V \subseteq \mathcal{T}$ holds. Each node $a \in V$ corresponds to a node $a' \in \mathcal{T}$,

2. for each $b \in V_{\text{D}}$, there should be one and only one $(Y, b) \in D$ for which it holds that $a \in Y$ iff there is a path $\langle a', n'_i, \dots, n'_j, b' \rangle$ in the task dependency graph where $n_i, \dots, n_j \notin V$.

3. for all paths $\langle a', n'_i, \cdots, n'_j, b' \rangle$ in the task dependency graph for which it holds that $n_i, \dots, n_j \notin V$ and $a, b \in V$, $a \prec b$ should hold.

4. for all $(a', b') \in E, b \notin V_{\text{Init}}$.

The cost function $W : \mathcal{T} \rightarrow \mathbb{N}$ associates a cost with each task to indicate the amount of work the task represents.

The cost or **length** of a path is be defined as the sum of the costs for each task on the path. Similarly, we define the **dag depth** T_∞ of a graph to be the highest cost for any path in the graph.

Note that, unlike the justification graph, the task dependency graph is not a hypergraph. Also, by definition, a task dependency graph cannot have any cycles, even if there are cycles in the corresponding justification graph.

The requirements for correspondence serve several purposes. The first requirement states that each node of the justification graph be represented by a separate tasks in the task dependency graph. We will call these tasks **match tasks**. To allow a more precise specification of the tasks carried out by a parser, we allow additional tasks with arbitrary dependencies to be inserted between two successive match tasks. The second and third requirement ensure correspondence between the task dependency graph and justification graph in this case.

In addition, the second requirement states that a match task may only depend on other items in correspondence to one and only one dependency in D. We do not allow a match task to depend on multiple derivations, because once an item is derived, it can already be used for the derivation of other items. If the same item is derived again, a parser can detect equality and terminate the execution of the corresponding thread. A parser based on the double dotted schema **DD** will typically produce many identical items. This is because the concatenation rules of **DD** allow the same item to be derived in multiple ways. For example, the item $a = [A \rightarrow \cdot BCD \cdot]$ can be derived from the items $b = [A \rightarrow \cdot B \cdot CD]$, $c = [A \rightarrow A \cdot B \cdot C]$, and $d = [A \rightarrow AB \cdot C \cdot]$, by concatenating in the order $((bc)d)$ and $(b(cd))$. Obviously, the number of possible identical derivations grows for larger righthand sides. Equality checking is therefore a crucial tool to reduce the number of items produced by Deltra. In the construction of a task dependency graph for Deltra, we simply pick the first derivation that was used during a sequential run. This choice will automatically eliminate any cycles that were present in the justification graph. Things are a little simpler for LinGO. A parsing system based on LinGO and **KD** will hardly ever yield multiple identical items. **KD** does not produce identical items like **DD**, as described before. In addition, the LinGO grammar has the property that different feature structures typically do not lead to the derivation of identical items. CaLi therefore omits the equality check and hence there will always be exactly one $(Y, b) \in D$ for each $b \in V_{\text{D}}$.

Finally, the fourth requirement disallows match tasks corresponding to initialization items in the justification graph to be the target of dependencies. Typically, initialization items are

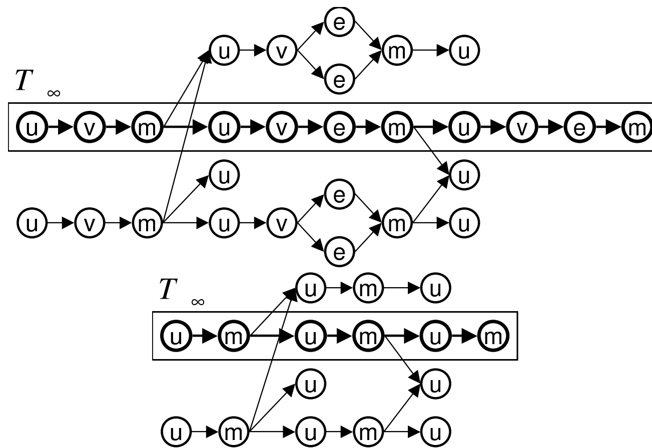


Figure 4.1 Example task dependency graph and its abbreviated version. In both graphs, a box is drawn around the critical path.

derived once and reused in each successive parse. Therefore, we do not want to include their derivation in the critical path analysis.

To fill in the details of the graph we have to look at the specific implementation of a chart parser. In Section 2.3, we introduced a general scheme for chart parsers called the unify–verify–match cycle. We define separate tasks for each of these operations. The match tasks correspond to the match step. We also introduce tasks corresponding to the unification and verification steps, including the equality check. The presence of these tasks will depend on the parser implementation. Each successful unification that results in a unique new item will result in the spawning of a match task. The number of failing unifications that follow a match task will partially depend on the filtering technique being used. We therefore need to consider the actual parsing behavior to determine the makeup of the task dependency graphs. An example of a task dependency graph is shown in Figure 4.1.

In the discussion below, we will represent the unification phase and the optional verification phase as a single task. The abbreviated version is also shown in Figure 4.1. Equality checks slightly complicate the analysis. In addition, equality checks can be accelerated considerably by using, for example, the quick check. We will therefore not consider the possibility of performing these tasks in parallel.[1]

We did not split match tasks into smaller units of execution, because they typically require much less time to complete than unification and verification tasks. Therefore, there is no real need to introduce fine-grained parallelism in this operation.

Although tasks are considered as atomic units of execution, the amount of work associated with a task can vary considerably. This especially holds for the equality check and unification tasks. A simple indicator of the cost involved in a unification task and the equality

[1] Note that CaLI does not implement equality checks.

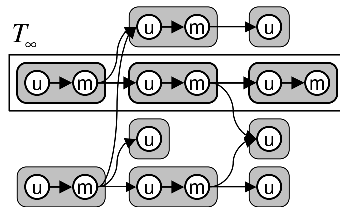


Figure 4.2 Type 1 task graph. Each unification task starts in a new thread. Threads are represented by the gray boxes.

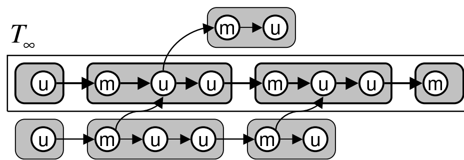


Figure 4.3 Type 2 task graph. Each match phase starts in a new thread.

check is to count the number of nodes visited during the operation. This measure cannot be used to indicate a cost for the match phase. However, since computing the matches takes up only a few percent of the total work, we can simply assign a small constant cost (typically zero) to each task.

4.3.2 Task Graphs

In most implementations of a parallel parser, tasks as presented above will not be the smallest schedulable units. Scheduling each type of task independently might be cumbersome and introduce unwanted inefficiencies. Therefore, a series of successive tasks will typically be grouped into a thread. A single thread can, for example, perform one complete iteration in the unify–verify–match cycle. Let us first define how we will allow tasks to be grouped into threads.

Definition 4.6 Given a task dependency graph $G = \langle \mathcal{T}, E \rangle$ we define a **thread grouping** for G to be a set of threads T, where each thread $t \in T$ represents a connected subgraph $\langle \mathcal{T}_t, E_t \rangle$ of the task dependency graph, and all tasks $n \in \mathcal{T}$ belong to exactly one thread. A **task graph** $G' = \langle \mathcal{T}, E' \rangle$ for G derived from a given thread grouping T is defined as a directed graph for which it is the case that:

1. For each $\langle \mathcal{T}_t, E_t \rangle \in T$, it holds that for each $a \in \mathcal{T}_t$ there is at most one edge $(a, b) \in E'$ where $b \in \mathcal{T}_t$; and, if $(a, b) \in E_t$ then there is a path $\langle a, n_i, \dots, n_j, b \rangle$ in G' for which holds that $n_i, \dots, n_j \in \mathcal{T}_t$.
2. For each $a, b \in \mathcal{T}$, where a and b belong to different threads, $(a, b) \in E'$ if and only if $(a, b) \in E$.

Requirement 1 states that there should be a single thread of execution within the thread that preserves the dependencies as defined by the task dependency graph. Since the task dependency graph is cycle free, this is always possible. Requirement 2 ensures that dependencies between tasks belonging to different threads are preserved.

We will consider two different task graphs where each thread represents a single iteration of the unify–verify–match cycle. The most obvious place to cut the cycle is at the matching step. This scheme puts each unification task at the start of a different thread and, hence, yields a very fine-grained distribution scheme. We will call these **type 1** task graphs. Figure 4.2 shows a type 1 task graph derived from the task precedence graph of Figure 4.1. Past research, however, has also focussed on schemes where a thread starts with a match step. We will call these **type 2** task graphs. Figure 4.3 shows the task graph that results from applying this scheme to the example task precedence graph.

As unification tasks always have at most one outgoing edge in the task dependency graph, a task precedence graph uniquely determines the corresponding type 1 task graph. This is not the case, however, for type 2 task graphs. The completion of a match task may spawn multiple unification tasks. Because of this, the order of execution of unification tasks in one thread is arbitrary. In addition, a unification task may depend on multiple match tasks. It can only be associated, however, with one thread. So the type 2 graph shown in Figure 4.3 shows only one of the possible instances of a type 2 task graph.

4.3.3 Metrics on the Task Graph

To derive the average available parallelism $\bar{P}$ from a task graph, we need to determine both the total work T_1 and the critical path length T_∞. The total work can simply be determined by summing the work of all tasks: $T_1 = \sum_{t \in \mathcal{T}} W(t)$. This value is the same for both type 1 and type 2 task graphs. Since there is only one type 1 task graph for each task precedence graph, the critical path length T_∞ of a type 1 graph is uniquely determined for each parse attempt.

This is not the case for type 2 graphs. Different instances of type 2 graphs can have different critical paths. A useful choice for T_∞ given a task precedence graph in this case is to take the worst case considering all possible instances of type 2 task graphs.

To find the worst case of all possible type 2 graphs, we will construct a graph of which the dag depth is equal to this worst case. Consider a thread on the critical path that starts with match task t_m. In a worst case scenario, the thread will include each unification task t_u for which there is an edge (t_m, t_u) in the task precedence graph *and* the last unification task executed leads to the next match task on the critical path. We can reflect this worst case in the desired graph by letting each match task be represented as a single node where its cost is the same as in the worst case scenario.

Definition 4.7 Given a simplified task precedence graph $P = \langle \mathcal{T}, E, W \rangle$, it holds for $T_{\text{wc}}(P) = \langle \mathcal{T}', E', W' \rangle$ that:

- For each match task $t_m \in \mathcal{T}$ there is a corresponding node $t'_m \in \mathcal{T}'$.

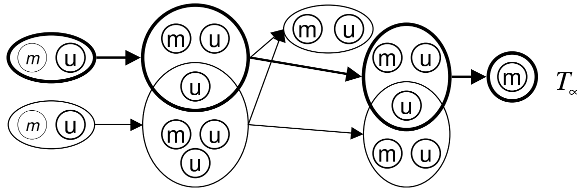


Figure 4.4 Utility graph used to compute the worst case critical path for type 2 graphs. The big circles represent the nodes. Each node encircles the tasks for which the cost is counted in the respective node.

- $W'(t'_m) = W(t_m) + \sum_{(t_m,t)\in E} W(t)$.
- $(t'_{m_1}, t'_{m_2}) \in E'$ if there exists a $t \in \mathcal{T}$ for which it holds that $(t_{m_1}, t) \in E$ and $(t, t_{m_2}) \in E$.

$\lhd$

Figure 4.4 shows the constructed graph corresponding to the previous presented example. Note that the cost for a unification task can be counted in more than one node. We therefore need to prove that no task will be counted in more than one node that resides on the critical path.

Theorem 4.8 *Given a task dependency graph $P = \langle \mathcal{T}, E\rangle$ that represents a strict parsing system. The dag depth of graph $T_{\mathrm{wc}}(P) = \langle \mathcal{T}', E', W'\rangle$ is equal to the maximum of the dag depths of all type 2 task graphs that can be derived from P.*

Proof. Obviously, because each match node in $T_{\mathrm{wc}}(P)$ represents the maximum possible cost for any instance, no single instance of a type 2 graph for P can yield a longer critical path than the dag depth of $T_{\mathrm{wc}}(P)$. What is left to prove is that no cost for any unification task is counted more than once.

From the definition of $T_{\mathrm{wc}}(P)$, the cost for a unification task $t_u \in \mathcal{T}$ is counted more than once if and only if there are two distinct tasks $t'_{m_1}, t'_{m_2} \in \mathcal{T}'$ on the critical path, corresponding to the tasks $t_{m_1}, t_{m_2} \in \mathcal{T}$, for which it holds that $(t_{m_1}, t_u) \in E$ and $(t_{m_2}, t_u) \in E$. By definition, if two match tasks t_{m_1} and t_{m_2} lead to the same unification task, $t_{m_1} \otimes t_{m_2}$ holds. Also, if t'_{m_1} and t'_{m_2} are both on the critical path, then there must be a path from t'_{m_1} to t'_{m_2}. But, this means there should also be a path from t_{m_1} to t_{m_2}. Hence, considering the correspondence between task precedence graphs and justification graphs, $t_{m_1} \prec t_{m_2}$ should hold. Given the requirement, though, that the parsing system corresponding to P be strict, $t_{m_1} \otimes t_{m_2}$ and $t_{m_1} \prec t_{m_2}$ cannot hold simultaneously. Hence, no unification task is counted twice in the critical path length. $\square$

4.3.4 Results

The task graph experiments were conducted for the LinGO grammar. We considered both type 1 and type 2 task graphs. For type 2 graphs we analyzed both an arbitrary graph, where

n	T_1	d	**Type 1** $\bar{P}$	**Type 2** $\bar{P}$	**WC Type 2** $\bar{P}$
1–5	21777	6	14	7	5
6–10	180838	13	57	22	12
11–20	1533983	18	284	81	17
21–30	2789718	21	452	119	22
31–40	3817997	23	560	156	24
41+	2594891	23	413	125	26
all	1014247	14	188	55	14

Table 4.1 Critical path analysis for type 1 and type 2 task graphs. For type 2 task graphs, both an arbitrary case and the worst case are shown.

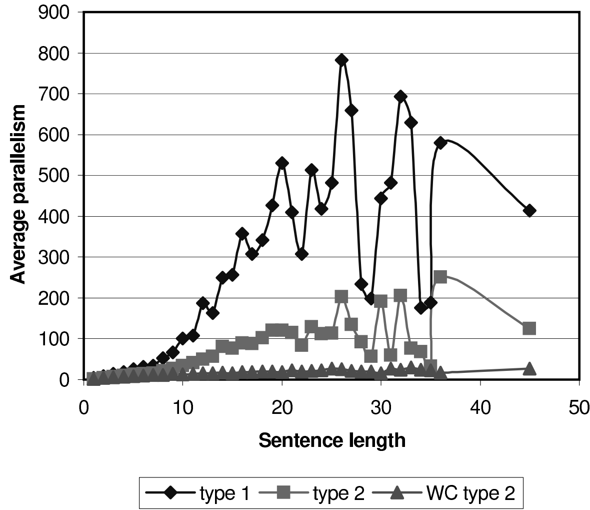
the dependencies were derived from the sequential execution, and the worst case graph. The cost for the unification and copy operations were derived by counting the number of nodes involved.

Table 4.1 shows the results for each type of task graph, averaged over all sentences in the fuse test set. It shows the total amount of work T_1 and the **derivation tree depth**, which is equal to the number of match tasks on the critical path. This measure will be used for the design of the parser in Chapter 7. Furthermore, for each type of task graph the average parallelism is shown.

The results show that the average parallelism in type 1 graphs is considerably larger than in type 2 graphs, even when the arbitrary case is considered. In general, the average parallelism seems to become considerable for sentences of 8 or more words. Thus to maximize the average parallelism one should allow each unification task to be scheduled independently. The average amount of parallelism obtained with the type 2 approach is often too small to allow this approach to be considered useful for the implementation of a parallel parser. The same results are reflected in Figure 4.5. We can therefore conclude that a viable approach to parallel parsing should be based on a type 1 approach to the distribution of work.

4.4 Parallelism and Parsing Schemata

The parsing schema that is used for parsing has an influence on the critical path length. Intuitively, one might expect that a bottom-up parsing schema will yield a smaller critical path than a top-down parsing schema. In this section, we will investigate the influence of a parsing schema on the critical path length. To eliminate the need to implement different parsers, we perform a more abstract analysis, as we will discuss next.



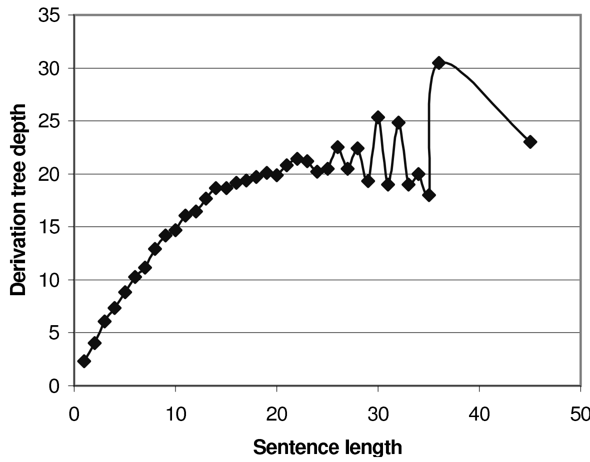


Figure 4.5 Average parallelism and average derivation tree depth for the respective justification graphs.

4.4.1 Justification Graph as Task Graph

A justification graph for one parsing schema can be transformed to a justification graph of another parsing schema by using the transformation steps discussed in Section 2.1. So using the justification graph as a basis to predict results of other parsing schemata eliminates the need to implement a parser for each schema that is subject of investigation. Although the justification graph is not as detailed as a task dependency graph—for example, it does not include failing unifications—it provides a good approximation, as we will argue next.

Profiling data on both CaLi and Deltra show that UNIFY is responsible for about 90% of the computation time in a typical parsing. In addition, almost 80% of this time is spend in the COPY function, which is called by UNIFY whenever a unification succeeds. This implies that successful unifications must make up for at least 72% of the parsing time. We can model this by assuming the parser uses a $O(1)$ oracle that can determine in advance whether a match will fail or succeed. In this case, the cost of parsing is mostly determined by the succeeding unifications (and the successive copying), as there is no need to process unifications if they are known to fail in advance. Using the filtering techniques, many recent parsers more or less approximate such an oracle.

When all tasks corresponding to filtered out unifications are eliminated from the task dependency graph, the resulting graph is already very similar to a justification graph. In fact, each thread of the corresponding type 1 task graph includes exactly one unification and match task[2] and corresponds to exactly one of the derived items in the justification graph. We can therefore define the cost of processing a node in the justification graph as the cost of the corresponding match tasks and its preceding unification task in the task dependency graph.

4.4.2 Base Parsing Schema

In order to conveniently transform a single justification graph to other graphs representing different parsing schema, we need a parsing schema that can easily be expressed in other parsing schemata by means of simple mappings. A useful taxonomy of parsing schemata is given in (Sikkel, 1993b). At the top of the taxonomy is a single parsing schema from which all parsing schemata we are interested in can be derived. It is a variant of the double dotted parsing schema, with deduction steps showing close resemblance to Earley. It is defined as follows.

Parsing Schema 4.9 (DD$_0$)

$$\mathcal{I}_{\mathrm{DD}_0} = \{[A \rightarrow \alpha \bullet \beta \bullet \gamma, i, j] \mid A \rightarrow \alpha\beta\gamma \in P \wedge 0 \le i \le j\}$$

$$\mathrm{D}^{\mathrm{Init}} = \{\vdash [A \rightarrow \alpha \bullet\bullet \gamma, j, j]\}$$

$$\mathrm{D}^{\mathrm{Scan}} = \{[A \rightarrow \alpha \bullet\bullet a\gamma, i, i], [a, i, i+1] \vdash [A \rightarrow \alpha \bullet a \bullet, i, i+1]\}$$

$$\mathrm{D}^{\mathrm{Complete}} = \{[A \rightarrow \alpha \bullet\bullet B\gamma, i, i], [B \rightarrow \bullet \beta \bullet, i, j] \vdash [A \rightarrow \alpha \bullet B \bullet, i, j]\}$$

$$\mathrm{D}^{\mathrm{Concatenate}} = \{[A \rightarrow \alpha \bullet \beta_1 \bullet \beta_2\gamma, i, j], [A \rightarrow \alpha\beta_1 \bullet \beta_2 \bullet \gamma, j, k][A \rightarrow \alpha \bullet \beta_1\beta_2 \bullet \gamma, i, k]\}$$

[2] We still assume the use of a simplified task dependency graph. Alternatively, there are tasks to perform equality or subsumption checks between the unification and match task.

	DD	**buE**	**buLC**	**E**
Length of derivation	1	0.84	0.84	1.8

Table 4.2 Derivation length of **buE**, **buLC**, and **E** relative to **DD**.

$$\mathrm{D}_{\mathrm{DD}_0} \quad = \quad \mathrm{D}^{\mathrm{Init}} \cup \mathrm{D}^{\mathrm{Scan}} \cup \mathrm{D}^{\mathrm{Complete}} \cup \mathrm{D}^{\mathrm{Concatenate}}$$

By using step contraction we can obtain, for example, a double dotted, a bottom-up Earley, and a Earley schema. It holds that $\mathbf{DD}_0 \stackrel{\mathbf{sc}}{\Longrightarrow} \mathbf{buE}$ and $\mathbf{DD}_0 \stackrel{\mathbf{sc}}{\Longrightarrow} \mathbf{DD} \stackrel{\mathbf{df}}{\Longrightarrow} \mathbf{E}$. This parsing scheme is quite inefficient. It results in a lot of redundant work. The only reason we use this parsing schemata is that it allows us to derive all desired justification graphs.

4.4.3 Measurements

To investigate the effect of the choice of parsing schema on the critical path length we transformed some of the justification graphs obtained from the Deltra parser. $\mathbf{DD}_0$ shows close resemblance to **DD**, which is used for Deltra. It was straightforward to generate output that reflects each of the steps produced by $\mathbf{DD}_0$.

To transform the base justification graph to a graph representing a different parsing schema, we use a set of transformation rules. First all deductions and items are transformed using simple rules. During this process we eliminate all nodes that resemble items that are not part of the new item domain. Finally, we eliminate all edges that do not connect valid items. These graph-based transformation rules were encoded in Prolog.

Table 4.2 shows the average increase of the critical path length of the derivations relative to the critical path lengths of the double dotted parser. The relative differences in length are averaged over all test sentences.

As expected, the bottom-up parsing schemata **DD**, **buE**, and **buLC** have a shorter critical path. On average, **E** results in a critical path of almost twice the length of the critical path of **DD**. Perhaps slightly surprising is that the single dotted parsing schemata **buE** and **buLC** yield a shorter critical path than **DD**. The double dotted schema **DD** allows multiple sections of the right-hand side of a rule to be processed in parallel, supposedly allowing for a shorter critical path length. However, **DD** is a more refined parsing schema than **buE** and **buLC**. Consider, for example, $\mathbf{DD}_0 \stackrel{\mathbf{sc}}{\Longrightarrow} \mathbf{buLC}$. By definition, $\mathrm{D}^{\epsilon}_{\mathbf{buLC}} \subseteq \mathrm{D}^{\epsilon}_{\mathbf{DD}}$, $\mathrm{D}^{\mathrm{LC}(a)}_{\mathbf{buLC}} \subseteq \mathrm{D}^{\mathrm{Init}}_{\mathbf{DD}}$, and $\mathrm{D}^{\mathrm{LC}(A)}_{\mathbf{buLC}} \subseteq \mathrm{D}^{\mathrm{Include}}_{\mathbf{DD}}$. However, for $\mathrm{D}^{\mathrm{Scan}}_{\mathbf{buLC}}$, an arbitrary deduction step

$$[A \rightarrow \alpha \bullet a\beta, i, j], [a, j, j+1] \vdash [A \rightarrow \alpha a \bullet \beta, i, j+1]$$

is emulated in **DD** by

$$[a, j, j+1] \vdash [A \rightarrow \alpha \bullet a \bullet \beta, j, j+1]$$

$$[A \rightarrow \bullet\alpha\bullet a\beta, i, j], [A \rightarrow \alpha\bullet a\bullet\beta, j, j+1] \vdash [A \rightarrow \bullet\alpha a\bullet\beta, i, j+1].$$

and similarly for $\mathrm{D}_{\mathbf{buLC}}^{\mathrm{Complete}}$, an arbitrary deduction step

$$[A \rightarrow \alpha\bullet B\beta, i, j], [B \rightarrow \gamma\bullet, j, k] \vdash [A \rightarrow \alpha B\bullet\beta, i, k]$$

is emulated in **DD** by

$$\begin{aligned}[B \rightarrow \bullet\gamma\bullet, j, k] &\vdash [A \rightarrow \alpha\bullet B\bullet\beta, j, k]\\ [A \rightarrow \bullet\alpha\bullet B\beta, i, j], [A \rightarrow \alpha\bullet B\bullet\beta, j, k] &\vdash [A \rightarrow \bullet\alpha B\bullet\beta, i, k].\end{aligned}$$

Obviously, in these cases, **DD** requires twice as many deductions to come to the same results. The net effect is a 16% reduction on the critical path length for **buE** and **buLC**.

Until now we considered the effect of the parsing schema on the critical path length. To determine the average parallelism, we should also consider total amount of work T_1. **DD** typically yields many more deductions and hence much more work than any of **E**, **buE**, or **buLC**. Therefore, computations for **DD** will typically yield the highest degree of average available parallelism. On the other hand, we do not want to increase the parallel slackness of a computation simply by introducing more work. Analogous to the work-first principle, as long as there is enough parallelism in the sequential version, the critical path has a minimal impact on the speedup. In this case we should aim at minimizing work, that is, keep T_1 close to T_{seq}. Introducing a parsing schema that yields a smaller critical path but more work is counterproductive in this case. However, when an Earley based parser has limited possibilities for parallelism, the results show that using a bottom-up parsing schema can reduce the critical path by about a factor of 2. So we can conclude that **DD** is not very useful for increasing the average available parallelism. Alternatively one could resort to the **OCYK** or **Rytter** parsing schemata in these cases.

There is one last observation, though, that is worth mentioning. Figure 4.6 shows a rough estimation of how the average amount of parallelism fluctuates during parsing for a **E** and **DD** parser. The **DD** parser shows a course that is typical for all bottom up parsers. The top-down parsing approach of **E** shows a much spikier course. This suggests that with **E** the chances that some processors temporarily run out of work are increased. The indications are not strong enough, though, to assume that this will actually be the case. It seems therefore that both **E** and **DD** yield sufficient parallelism to be effectively used for parallel parsing.

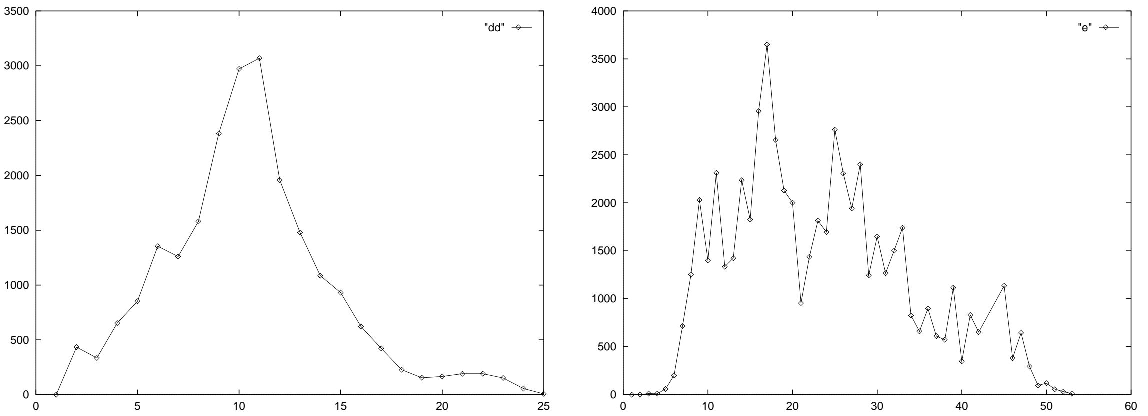


Figure 4.6 Two exemplary courses of the number of deductions per step during parsing. The left graph displays a typical course of the number of deductions per step for a double dotted parser, the right graph shows a typical course for an Earley type parser. The x-axis represents the number of processors in use. The y-axis represents the number of outstanding tasks.

Chapter 5

Communication in Parallel Parsing

In the previous chapter, we explored the amount of parallelism in parsing computations. In this chapter, we will investigate the amount of locality inherent in parsing. The amount of locality determines to what extent communication can be minimized when distributing the work involved with parsing amongst processors.

We show that a straightforward distribution of work amongst processors can easily lead to intolerable amounts of communication. We also show that choosing the right method for distributing work can reduce the total amount of communication to manageable quantities.

5.1 Introduction

Communication can be a bottleneck in obtaining sufficient speedup in parallel parsing. When there is too much communication between processors, the network, or the memory bus in case of shared-memory systems, can become saturated, causing significant delays for all processors. As we will show, the total amount of communication involved in parsing can exceed four times the total size of the unification-based structures. The unstructured nature of parsing computations can make it hard to find a distribution that minimizes communication and still divides the work evenly amongst the processors. In this chapter, we will investigate the bounds on the total and minimum communication for parsing and techniques to minimize the communication. We will make use of the theory presented in Section 3.3 for this analysis.

In the next section, we will introduce a basic model for the communication involved in the parsing process. In Section 5.3, we will present heuristics that can be used to minimize the total communication in a dynamic setting. Finally, in Section 5.4, we will present an analysis of the communication involved in parsing and an evaluation of the various heuristics.

5.2 Communication Model

The task dependency graph introduced in Section 4.3.1 provides a good starting point for the definition of a communication model for parsing. The task dependency graph incorporates the same tasks, and costs for executing the tasks, that are required for analyzing communication. In addition, the edges of the graph, indicating execution dependencies, also indicate the flow of data and, hence, communication between tasks.

Definition 5.1 Given a task dependency graph $T = \langle \mathcal{T}, E, W \rangle$ and a function $S_{\mathcal{T}} : \mathcal{T} \rightarrow Z$, where $S_{\mathcal{T}}$ assigns a measure of the size of an item associated with each match task

in $\mathcal{T}$ and 0 for all other tasks in $\mathcal{T}$. The **communication graph** corresponding to T and function $S_{\mathcal{T}}$ is a quadruple $\langle \mathcal{T}, E', W, S \rangle$ where

1. $(a, b) \in E'$ iff $(a, b) \in E$ or $(b, a) \in E$,
2. S is a function that assigns a communication cost to each edge $(a, b) \in E'$ where $S((a, b)) = S_{\mathcal{T}}(a) + S_{\mathcal{T}}(b)$.

The **maximum total communication** of a communication graph is defined as the sum of all data communicated, or $\sum_{e \in E'} S(e)$. $\lhd$

Note that because there are no edges in E that connect two match tasks, for all edges $(a, b) \in E$ either $S_{\mathcal{T}}(a) = 0$ or $S_{\mathcal{T}}(b) = 0$. The message size associated with each edge of the communication graph therefore corresponds to the size of exactly one item.

The match tasks are designated to be the owner of the items, whereas the unification tasks execute the work. An advantage of this model is that it allows complete freedom in assigning data and computations to processors independently.[1] Consider an operation where two items are successfully unified into a new item. The two input items and the result item are all associated with a different match task, and can therefore all be assigned to different processors. The corresponding unification task will typically be executed on one of the processors assigned to any of the respective match tasks, but can also be assigned to yet another processor.

Note that in the transformation of a task dependency graph to a communication graph, all task dependencies are lost. This means that in finding the optimal partitioning we disregard these constraints. Experiments on graphs obtained from our parser, however, indicate that this has minimal to no impact on the outcome of the analysis.

The fact that we associated the data size with the individual edges already indicates we aim at using the edge cut for measuring communication. This corresponds to measuring the amount of communication based on the network model, where the processors do not store items received by other processors. That is, it takes into account that an item needs to be transferred from one processor to the other each time it is required by this processor.

This model introduces some inaccuracies when used to analyze the communication on a shared-memory architecture. With shared-memory architectures, communication between processors takes place through shared memory. Once an item is “transferred” from one processor to another, it may remain in cache until it is needed next time. In addition, structure sharing can increase the chance that parts of the graph are already in cache. Finally, in case of premature failure of a unification often only a part of the graph needs to be communicated. To take all these aspects into account would require an analysis to be performed at the level of individual nodes, or at least subgraphs, of the feature structures, rather than complete graphs. As the analysis of the fuse test suite using complete graphs already produces communication graphs of over 100,000 nodes, enlarging the graphs even further is not desirable.

Nevertheless, basing the analysis on the network model can still give a good indication of the communication patterns involved in parallel parsing. When an item needs to be transferred

[1] We assume that match tasks are assigned a zero cost for the amount of work; see the discussion in Section 4.3.1.

from one processor to the other for the first time, we know it cannot be in the cache of the destination processor. An amount of communication, according to the network model, can therefore give an indication of the number of coherency misses.

5.3 Grouping Heuristics

A random distribution of tasks amongst processors can be expected to yield considerably more communication than strictly necessary. Since memory bandwidth is scarce, a parallel parser should incorporate some mechanism to reduce communication between processors. Because the communication graph of a parse attempt is not known beforehand, domain decomposition tools are not useful in solving this problem, A practical solution to minimizing communication is to group related tasks on one processor. This ensures that the exchange of data between tasks belonging to the same group does not yield any communication.

5.3.1 Reducing the Upper bound of Communication

We define a grouping of tasks in a communication graph in a similar fashion as the grouping of tasks in a task dependency graph, as presented in Definition 4.6.

Definition 5.2 Given a communication graph $C = \langle \mathcal{T}, E, W, S \rangle$, we define a **grouping** to be a set $\mathcal{G}$ of sets of tasks, such that each task in $\mathcal{T}$ is contained in exactly one of the sets in $\mathcal{G}$. The grouping $\mathcal{G}$ of a communication graph $C = \langle \mathcal{T}, E, W, S \rangle$, is said to yield a communication graph $\langle \mathcal{T}', E', W', S' \rangle$ for which it is the case that

1. For each $\mathcal{T}_a \in \mathcal{G}$ there is a corresponding $t_a \in \mathcal{T}'$.
2. For each $t_a \in \mathcal{T}'$ with corresponding $\mathcal{T}_a \in \mathcal{G}$, $W'(t_a) = \sum_{t \in \mathcal{T}_a} W(t)$.
3. $(t_a, t_b) \in E'$ if and only if for the corresponding groups $\mathcal{T}_a \in \mathcal{G}$ and $\mathcal{T}_b \in \mathcal{G}$ there exists a $t_1 \in \mathcal{T}_a$ and a $t_2 \in \mathcal{T}_b$ for which $(t_1, t_2) \in E$.
4. For each $(t_a, t_b) \in E'$ and corresponding $\mathcal{T}_a \in \mathcal{G}$ and $\mathcal{T}_b \in \mathcal{G}$, $S'((t_a, t_b)) = \sum_{\forall s_a \in \mathcal{T}_a \forall s_b \in \mathcal{T}_b \cdot (s_a, s_b) \in E} S((s_a, s_b))$.

A **grouping heuristic** is defined as a function that produces a grouping from a communication graph. $\lhd$

From Condition 4 it follows that the maximum total communication of the communication graph resulting from applying the grouping heuristic is smaller or equal to the maximum total communication of the original graph. All communication between tasks belonging to the same group is never counted. Grouping tasks such that the number of edges between tasks within one group is increased therefore decreases the total communication.

As we saw in Chapter 4, however, too much grouping of tasks can make it impossible to obtain a balanced partitioning for a larger number of processors. The trick is therefore to find

a grouping heuristic that reduces the communication as much as possible, while allowing sufficient room for obtaining a well-balanced partitioning.

The results of the analysis of grouping heuristics in this chapter are used in Chapter 7 for the design of a parallel parser. In this design, grouping heuristics are used as scheduling guidelines, rather than a strict requirement. This changes the meaning of the well-balancedness of a partitioning, because balance can always be restored by deviating from the guidelines. However, it can be expected that deviating from the guidelines by moving data back and forth between processors can incur additional communication overhead. In this scenario, the balance can therefore be seen as the level of guarantee that a grouping heuristic can accomplish the reduction in communication.

5.3.2 Grouping Heuristics

An important requirement of grouping heuristics is that they be fast and implementable. By fast we mean that they should not incur significant overhead to compute. With implementable we mean that the group in which a task is to be put should be determinable at run-time. In this section, we present three different grouping heuristics that meet these requirements.

Rule-based Distribution The **rule-based** grouping heuristic was inspired by Yonezawa and Oshawa's (1994) approach to distributing work based on the non-terminals of the production rules. Instead of grouping tasks based on the non-terminal level, however, we took the less fine-grained approach of grouping tasks per grammar rule.

The rule-based grouping heuristic states that all items which are associated with the same grammar rule be stored and derived on the same processor. That is, given a parsing system $\mathbb{P}$ for a grammar $G = \langle N, \Sigma, P, S \rangle$, we define each unification task and match task corresponding to, respectively, the computation and storage of an item $[A \rightarrow \alpha \bullet \beta \bullet \gamma, i, j]$ to be a member of $\mathcal{T}_{A \rightarrow \alpha\beta\gamma}$. Hence, each $p \in P$ is associated with a group $\mathcal{T}_p$. In addition, each match task corresponding to a lexical entry is associated with its own group. The maximum number of groups is limited by the number of rules of the grammar plus the total number of lexical entries involved in a parse.

The idea of grouping items per grammar rule is that deduction rules of parsing schemata often include items which are associated with the same rule. As a result, it is guaranteed that all items involved in such a derivation are always on the same processor.

Obviously the effectiveness of this kind of grouping relies on the particular grammar and parsing schema being used. For example, all items involved in a concatenation of the double dotted parsing schema **DD** are associated with the same rule. So, in this case, the rule-based heuristic will allow all concatenations to be executed without incurring any communication. For the key-driven parsing schema **KD**, this grouping mainly has the effect of ensuring that the result item is stored on the same processor that executes the corresponding unification task. All other savings in this case are largely coincidental.

Tabular Chart Cell-based Distribution The **tabular chart cell-based** grouping heuristic is based on the distribution that is used in parallel tabular chart parsers. With this heuristic, all items that cover the same part of the input string are stored and derived on the same processor. That is, given a parsing system $\mathbb{P}$ and corresponding task dependency graph, we define each unification task and match task corresponding to respectively the computation and storage of an item $[A \rightarrow \alpha \bullet \beta \bullet \gamma, i, j]$ to be a member of $\mathcal{T}_{i,j}$. Hence, we define a group $\mathcal{T}_{i,j}$ for each $0 \leq i \leq j \leq n$, where n is the length of the input string. The maximum number of groups is limited by $\frac{1}{2}(n^2 + n)$, which is the number of cells in the tabular chart.

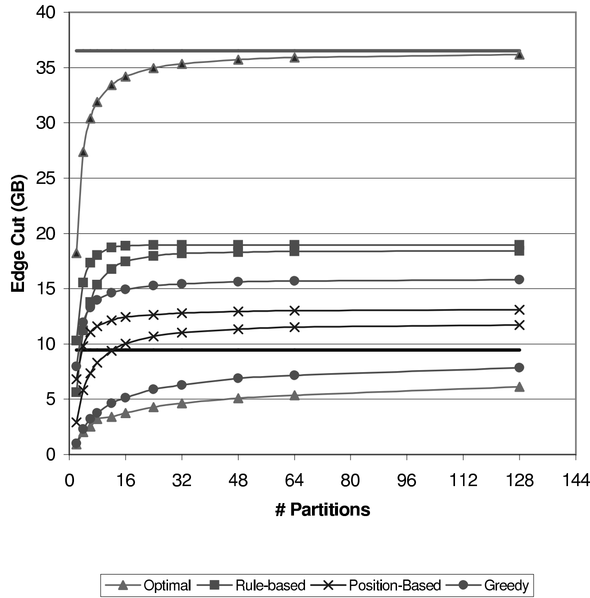
The idea behind this approach is that many parsing schemata have the equivalent of an inclusion rule. With this derivation rule, there is one item on each side of the production, where the item on the left-hand side is combined with a grammar rule to produce the item on the right-hand side. Since both items always cover the same part of the input string, they are always associated with the same group. Assuming that all processors have access to a private copy of the grammar, all inclusion rules can be completed without yielding any communication. Both **DD** and **KD** have an inclusion rule and can be expected to benefit from this approach.

Greedy The **greedy** heuristic is based on the idea of using a distributed chart for parallel parsing. Basically, instead of distributing items according to some characteristic, the tasks are executed on the processor that happens to store the data involved in the operation. Firstly, each unification task is executed on the processor associated with any of the match tasks providing the input items. The definition of the grouping heuristic explicitly requires a task to be put in a single group, and not a collection of possible groups. We must therefore choose in which group we put the unification task if it has more than one input match tasks. In the remainder of this chapter we assume this selection is done at random. Secondly, a match task resulting from an inclusion rule is associated with the same processor as the corresponding unification operation. To have some degree of freedom for load balancing, we allow all other match tasks to be associated with an arbitrary processor.

The greedy heuristic imposes less restrictions than the tabular chart cell-based approach. As a consequence, the number of nodes of the resulting communication graph is still relatively large, providing more opportunities for load balancing. The greedy heuristic still shares the benefit with the tabular chart cell-based approach of ensuring that inclusion rules can be applied free of communication.

5.4 Evaluation

For each of the grouping heuristics presented in the previous section, we measured the edge cut for two different distribution techniques: an approximation of the optimal partitioning and a random partitioning. We use the random distribution technique as an indication of the average case where no effort is made to optimize a partitioning. The random partitioning is computed by assigning each node in the communication graph to one of the given number of partitions at random. The approximation of the optimal partitioning is obtained from passing the communication graph to a partitioning algorithm. There are several libraries



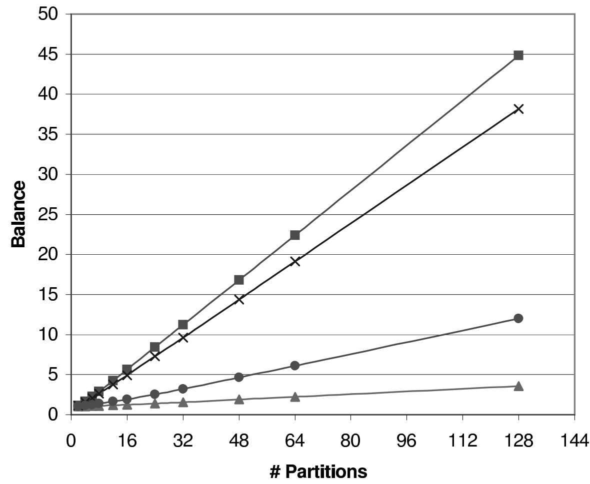


Figure 5.1 Minimum and random edge cut and minimum balance for various distribution types.

Distribution type	avg. # nodes	avg. # edges
Full graph	6604	16225
Rule-based	56	203
Tabular chart cell-based	73	188
Greedy	1893	6804

Table 5.1 Statistics of graphs resulting various grouping heuristics.

with partitioning algorithms that can give reasonable to good approximations. We used the METIS tool set, developed at the University of Minnesota (Karypis and Kumar, 1995). METIS produces good quality partitionings, also in comparison with other available tools (Karypis and Kumar, 1999).

The results were obtained using the first 1800 sentences of the fuse test set in combination with the LinGO grammar. Computational constraints, especially when finding the optimal partitionings for large sentences, made obtaining results for larger sentences impractical or even impossible. In addition, we omitted failing match tasks from the communication graph. A failing match task indicates a successful filtering of a unification task. As over 90% of the unification tasks can be filtered, this greatly reduces the number of nodes. We can assume that the effect of this simplification on the measurements is minimal. Firstly, in Section 4.3.1 we argued that the computational cost of the match tasks can be neglected. Omitting these tasks therefore has a minimal influence on load balancing. Secondly, to filter a unification task, it is not necessary to communicate the typed feature structures of the input graphs. Since the space required for items alone is only a fraction of the space required for feature structures, we can safely assume that omitting the respective match tasks also has a minimal influence on the volume of the communicated data.

Figure 5.1 shows the edge cuts resulting from the optimal and random partitionings of the communication graph and various grouping heuristics, summed over all sentences. The graph also includes two lines indicating respectively the total size of the items used in parsing—including feature structures—and the maximum total communication for the full communication graph. The former is included to give an idea of how the total volume of the communication relates to the volume of the data involved in the computation.

As can be seen, the use of the random partitioning for the communication graph quickly leads to the maximal possible communication. As could be expected, the near optimal and random partitionings of the communication graph delimit the edge cut size obtained with any of the grouping heuristics.

The rule-based heuristic is outperformed by the tabular chart cell-based and greedy heuristic on all counts. The rule based distribution produces the least balanced partitionings and produces partitions yielding a relatively large communication volume.

The minimal edge cut of the greedy heuristic is considerably better than the minimal edge cut produced by the other heuristics. The greedy heuristic only yields a slightly larger edge cut for random partitionings compared to the tabular chart cell-based approach. The greedy heuristic also yields more well-balanced distributions. From Table 5.1 it is evident that

the small number of nodes in the communication graphs produced by the rule-based and tabular chart cell-based heuristics prevent these heuristics from yielding a well-balanced partitioning in case of a larger number of processors.

In conclusion, our experiments showed that the use of grouping heuristics can considerably reduce the amount of communication. It seems that the greedy heuristic allows for partitionings that produce near-optimal edge cuts, while considerably limiting the worst-case communication. In addition, the resulting partitionings are fairly well-balanced, compared to the partitionings resulting from the other heuristics. We therefore can conclude that the greedy heuristic is useful in effectively reducing communication for a parallel parser based on the English LinGO grammar.

We note that for similar measurements on the Deltra grammar, the rule-based heuristic produced considerably better results. This can be explained by the use of the **DD** parsing schema in combination with the Deltra grammar. All items involved in the concatenate deduction rule of **DD** are always associated with the same rule. As a result, there are many deduction rule applications that can be completely processed with the information stored within a single group. Measurements show that this heuristic can reduce the communication for Deltra by roughly 60%. In addition, since the Deltra grammar contains a large number of grammar rules, the heuristic can produce reasonably well-balanced partitionings.

Chapter 6

Efficient Thread-safe Unification

Both in terms of speed and memory consumption, graph unification remains the most expensive component in unification-based grammar parsing. In this chapter, we will present two techniques to reduce the memory usage of such unification algorithms, with a minimal impact on execution times. In addition, the proposed algorithms are thread-safe. This allows them to be used in a parallel parser setup that exploits concurrent unification. Most of the discussion in this section will focus on the LinGO grammar. However, we will also present results obtained for the Deltra grammar. Parts of this chapter have been published in (van Lohuizen, 2001a) and (van Lohuizen, 2000).

6.1 Introduction

In Section 2.2.3, we presented Wroblewski's and Tomabechi's algorithms as efficient solutions to unification for natural language parsing. Their good performance is mainly obtained by reducing or eliminating the amount of superfluous copying. In order to avoid superfluous copying, these algorithms incorporate control data in the graphs. This has several drawbacks, as we will discuss next.

Memory Consumption To achieve the goal of eliminating superfluous copying, the aforementioned algorithms include administrative fields—which we will call **scratch fields**—in the node structure. These fields do not contribute to the definition of the graph, but are used to efficiently guide the unification and copying process. Before a graph is used in unification, or after a result graph has been copied, these fields just take up space. This is undesirable, because memory usage is of great concern in many unification-based grammar parsers. This problem is especially of concern in Tomabechi's algorithm, as for typical applications the scratch fields can make up from 40% to over 80% of the total memory required to store a node.

In the ideal case, scratch fields would be stored in a separate buffer allowing them to be reused for each unification. The size of such a buffer would be proportional to the maximum number of nodes that are involved in a single unification. A straightforward approach to separate the scratch fields from the nodes is to use a hash table to associate scratch structures with the addresses of nodes. However, this binding mechanism, as any other binding mechanism, will inevitably incur some overhead.

Nevertheless, considering the difference in speed between processors and memory, reducing the memory footprint may compensate for the loss of performance to some extent. Although separating the scratch fields can reduce memory usage considerably, it does not reduce

the amount of data involved in a single unification. Nevertheless, because nodes without scratch fields are smaller, storing and loading nodes to and from memory will be faster. In addition, because scratch fields are reused, there is a high probability that they will remain in cache. The total traffic between cache and memory is therefore most likely reduced. As the difference in speed between processor and memory continues to grow, caching is an important consideration (Ghosh et al., 1997).[1] In Sections 6.2 and 6.3, we will present two different binding techniques, respectively.

Concurrent Unification In Section 2.2.4, we presented two possible setups to exploit parallelism centered around the unification algorithm: parallel and concurrent unification. We believe that as long as the number of unification operations in one parse is large, it is preferable to choose concurrent unification. Especially when a large number of unifications terminate quickly (e.g. due to failure), the overhead incurred by the fine-grained parallelism obtained with parallel unification can be considerable.

With concurrent unification, a graph can be involved in multiple unifications simultaneously. This suggests that in order for concurrent unification to work, the input graphs need to be read only. However, including scratch fields in the node structure, as is done by Tomabechi's and Wroblewski's algorithms, thwarts the implementation of concurrent unification. Different processors will need to write different values in such scratch fields, inevitably causing conflicts. One way to solve this problem is to disallow a single graph to be used in multiple unification operations simultaneously. In Chapter 4, however, we saw that restricting the order of evaluation can increase the critical path and can considerably reduce the average available parallelism. Another solution is to duplicate the scratch fields in the nodes for each processor. This, however, will enlarge the node size even further. In other words, Tomabechi's and Wroblewski's algorithms are not well suited for concurrent unification.

6.2 Separating Scratch Fields

The key to the solution of all of the issues mentioned above is to separate the scratch fields from the fields that actually make up the definition of the graph. We have taken the version of Tomabechi's quasi-destructive graph unification algorithm presented in Section 2.2.3 as the starting point. With this algorithm, adapted for typed feature structures, a node structure comprises a type field, an arc list, a forward pointer, a generation counter, a new type field, and a comp arc list. Basically, the type and arc list fields suffice to describe the structure of the graph. The other fields can be considered scratch fields. Instead of including the scratch fields in the node structure, we put them in a structure we will call the **scratch buffer**. Instead of having a scratch buffer for each node, we maintain a limited set of these buffers. Each time we perform some operation on a set of graphs, we associate a buffer with each of the nodes of each graph. After the operation completes, and a possible result graph has been copied, all scratch buffers can be reclaimed and reused in successive operations.

[1] One might argue that most scratch fields that are written out need not be initialized, and hence do not take up bandwidth. However, cache lines are typically larger than a field, meaning that a single change in the node can cause the entire node (or more) to be written to memory.

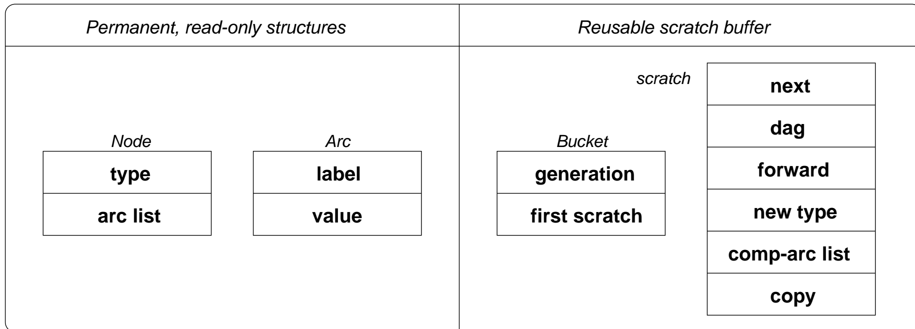


Figure 6.1 Node and Arc structures and the reusable scratch buffer for the hashing technique.

Obviously, since typically only a limited number of graphs is involved in one operation, we save a considerable amount of memory.

Having separated the scratch fields from the actual graph, we need a mechanism to associate such scratch buffers to nodes during unification and copying. The most straightforward method to accomplish this is to use a hash table. We can derive a hash value from the node's address. The hash value is used as an index in an array of buckets. Each bucket may contain multiple scratch buffers. In order to identify the correct buffer, we include the node's reference as a field (called dag) in its associated buffer. We also include a next field in the buffer to allow a linked list of buffers to be associated with a bucket. The resulting data structures are shown in Figure 6.1.

The basic mechanism for binding the scratch buffers to the nodes is incorporated in the DEREFERENCE$_{\text{hash}}$ function, which is shown in Figure 6.2. Scratch buffers are associated with nodes on demand. This avoids the overhead of traversing over all nodes involved in an operation beforehand. If the scratch buffer is found, the forward links are followed. Otherwise a new scratch buffer is initialized and added to the bucket. HASH derives a bucket location from the node's address.

All buckets can be invalidated by increasing a generation counter, analogous to the generation mechanism in Tomabechi's algorithm. Because DEREFERENCE$_{\text{hash}}$ ensures that all buffers in the list will be of the same generation, we only need to check for a valid generation once for every bucket. Scratch buffers can also be reclaimed efficiently. All scratch buffers are stored in an array. We can reclaim all buffers simply by setting an index pointing to the first free scratch buffer to the start of the array.

The added complexity of the hash table can almost entirely be hidden in DEREFERENCE$_{\text{hash}}$. The UNIFY and COPY function mainly need to be adapted to deal with the fact that the information for a node is now stored in two structures: the scratch buffer and the node itself. We let DEREFERENCE$_{\text{hash}}$ return a scratch buffer rather than just a scratch node. This prevents UNIFY and COPY from having to perform additional lookups. Note that a reference to the dereferenced node is included in the scratch buffer; hence, we do not have to return it explicitly.

```
DEREFERENCE_hash(dg)
  1. bucket ← table[HASH(dg)]
  2. if bucket.generation = generation then
       for each scratch in bucket.buffers do
         if scratch.dag = dg then
           2.1. while scratch.forward do
                    scratch ← scratch.forward
           2.2. return scratch
  3. else
       3.1. bucket.generation ← generation
       3.2. bucket.buffers ← nil
  4. Allocate next free scratch buffer scratch and add to bucket.
  5. scratch.dag ← dg
  6. scratch.newType ← dg.type
  7. Reset all other fields of scratch.
  8. return scratch
```

Figure 6.2 Dereference with intermediate hashing table

Once a graph is created, it is never altered again. This means that it will be safe to share such a graph using concurrent unification. The only requirement for thread-safe operation is that each processor use its own set of scratch buffers.

6.3 The Indexing Technique

Using hashing to associate scratch buffers with nodes can incur a considerable overhead. In this section, we will propose an alternative technique to associate scratch buffers with nodes. With the presented technique we aim to eliminate the overhead incurred by hashing. The technique also has some other advantages, as we will point out.

With the hashing technique introduced in the previous section, multiple nodes may map to the same bucket. This means we potentially have to proceed over a large list of scratch buffers before we find the one we actually need. We can avoid this overhead by assigning all nodes involved in an operation with a unique index into an array of scratch buffers. Basically, for each graph, we assign each node a unique index. We assign 0 to the root of the graph. The other nodes we label with increasing values in a depth-first manner.

Different graphs typically share the same indexes. Since unification involves (at least) two graphs, we need to ensure that two nodes will not be assigned to the same scratch buffer. We solve this by assigning an additional offset to each graph involved in the operation. These offsets can simply be computed by cumulatively adding the number of nodes, and hence the number of reserved scratch buffers, of all graphs.[2]

[2]To efficiently determine the number of required scratch buffers for a graph, we simply associate this number with each graph at the time of its creation. This information is available after copying without any additional cost.

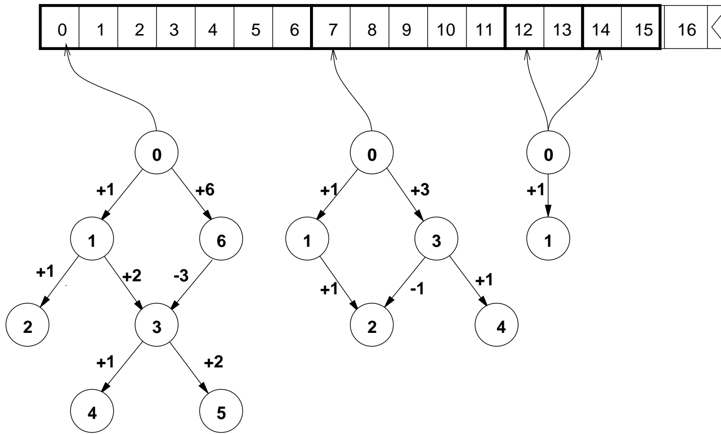


Figure 6.3 The mechanism associating index numbers with nodes.

Figure 6.3 shows an example of how unique scratch buffers can be assigned to all nodes of different graphs. The third graph is an example of how multiple copies of one graph can be represented by simply assigning multiple blocks of scratch buffers. We will discuss the use of this later in this section.

The offset that is associated with each node needs to be included in the graph. As can be seen in Figure 6.3, this information can be included either in the arc or the node structure. The nodes in this figure are associated with absolute offsets. The arcs are associated with difference in offsets between parent and child node. Either way of incorporating the offset data suffices to derive the appropriate index. Because it will slightly simplify the discussion of structure sharing later in this chapter, we will assume the data is stored in the form of relative offsets associated with arcs.

6.3.1 The Indexing Algorithm

The scratch buffers are similar to those used by the hashing technique. We do not need the next and dag field. We add a field for the generation. In addition, the references to other nodes (in forward, copy, and the comp_arc list) need to include the index of the respective node. Obviously, we do not use a bucket.

The resulting unification algorithm is shown in Figure 6.4. It is very similar to the algorithm presented in Section 2.2.3, but incorporates the necessary changes to implement the indexing technique. The node references passed to UNIFY1$_{\mathsf{idx}}$ are now a tuple, containing a reference to the node itself and the node's index. The index is used to locate the node's associated scratch buffer in the array sba. The indexes of the children of the node can be derived by adding the offsets associated with the respective arcs.

Whenever a reference to a node is recorded in a field of the scratch buffer (forward, copy, or in the comp_arc list), we need to record the index as well. When such a reference is needed

UNIFY$_{\mathsf{idx}}$(dg1, dg2)

1. *Allocate offsets* left_offset *and* right_offset *in the scratch buffer for graphs* dg1 *and* dg2, *respectively.*
2. **if** UNIFY1$_{\mathsf{idx}}$((dg1, left_offset), (dg2, right_offset))[a] **then**
   - 2.1. (copy, n) ← COPY$_{\mathsf{idx}}$((dg1, left_offset), 0)
   - 2.2. *Increase the generation counter.*
   - 2.3. **return** copy
3. **else**
   - 3.1. *Increase the generation counter.*
   - 3.2. **return** *nil*

UNIFY1$_{\mathsf{idx}}$(ref_in1, ref_in2)

1. (dg1, idx1) ← DEREFERENCE$_{\mathsf{idx}}$(ref_in1)
2. (dg2, idx2) ← DEREFERENCE$_{\mathsf{idx}}$(ref_in2)
3. **if** idx1 = idx2[b] **then**
   - 3.1. **return** *true*
4. sba[idx1].newType ← sba[idx1].newType $\sqcap$ sba[idx2].newType
5. **if** sba[idx1].newType = $\top$ **then**
   - 5.1. **return** *false*
6. **else**
   - 6.1. **if not** MAKEWELLFORMED((dg1, idx1)) **then**
     return *false*
   - 6.2. (dg1, idx1) ← DEREFERENCE$_{\mathsf{idx}}$((dg1, idx1))
7. FORWARD((dg2, idx2), (dg1, idx1))
8. **if** *Any of the nodes has arcs* **then**
   - 8.1. shared ← INTERSECTARCS((dg1, idx1), (dg2, idx2))
   - 8.2. **for each** ((_, r1), (_, r2)) **in** shared **do**
     UNIFY1$_{\mathsf{idx}}$(r1, r2)
   - 8.3. new ← COMPLEMENTARCS((dg1, idx1), (dg2, idx2))
   - 8.4. **for each** arc **in** new **do**
     Push arc *to* sba[idx1].comp_arcs
9. **return** *true*

FORWARD((dg1, idx1), (dg2, idx2))
  sba[idx1].forward ← (dg2, idx2)

ABSARC((label, (dg, offset)), parent_idx)
  return (label, (dg, parent_idx + offset))

[a] Note that if a node other than the root node is passed to UNIFY1$_{\mathsf{idx}}$, the offset needs to be adjusted accordingly.
[b] A node is uniquely identified by its index.

Figure 6.4 The unification algorithm.

DEREFERENCE$_{idx}$((dg, idx))

1. **if** sba[idx].generation $\neq$ generation **then**
  1.1. sba[idx].generation $\leftarrow$ generation
  1.2. sba[idx].newType $\leftarrow$ dg.type
  1.3. *Reset all other fields of* sba[idx]
2. **else**
  2.1. **while** sba[idx] $\neq$ *nil* **do**
      (dg, idx) $\leftarrow$ sba[idx].forward
3. **return** (dg, idx)

Figure 6.5 Dereferece for the indexing technique

in a future reference, the link to the parent is lost. The index of this reference can in this case no longer be determined by adding an offset to its parents index.

The function INTERSECTARCS determines the set of arcs with common labels for the input nodes. The function COMPLEMENTARCS returns the arcs with labels that exist in (dg2, idx2), but not in (dg1, idx1). Both functions use ABSARC to convert the relative offsets stored in the arcs to absolute indexes. The arcs stored in the comp_arcs field of the scratch buffer do not have to be converted, as these arcs already store an absolute index.

DEREFERENCE$_{idx}$ and FORWARD also have to take the additional index into account. DEREFERENCE$_{idx}$ uses a generation mechanism similar to that of Tomabechi to invalidate the contents of the buffer. The functions DEREFERENCE$_{idx}$, and FORWARD and ABSARC are shown in Figure 6.5 and 6.4, respectively.

MAKEWELLFORMED is defined as in Section 2.2.3, with the following changes. Before calling UNIFY1$_{idx}$, it allocates a block of scratch buffers for the nodes in the constraint, starting at the first free scratch buffer in sda.

The changes to the copy function are slightly more complicated, because we need to compute the offsets for the copied arcs. We keep track of free index numbers using a global variable free_idx. Each time a node is copied, we assign it the number stored in free_idx and increment free_idx accordingly. We do not need to reserve a new index position in case of a reentrancy (line 2.1). COPY$_{idx}$ returns a reference to the copied node and its offset in the result graph. At line 8.2, these indexes are converted to the offsets as we store them in arcs. The offset is computed simply by subtracting the index of the parent node from the index of the child node. Note that, for clarity, we have omitted any cycle checks.

The indexing technique has an additional advantage over Tomabechi's algorithm and the hashing variant of this algorithm presented in the previous section. The index of a node uniquely identifies this node during the course of a unification. This means we can represent multiple unique instances of this node simply by assigning each instance a different index. This property can be exploited, for example, when interleaving the unification of type constraints in MAKEWELLFORMED. It is possible that the same type constraint is applied more than once during a single unification. Using Tomabechi's algorithm, each instance of a constraint should be represented by a different copy, because each instance will typically hold different values in the scratch fields. Using the presented indexing technique,

$\text{COPY}_{\mathsf{idx}}((\mathsf{dg}, \mathsf{idx}))$

1. $\mathsf{free_idx} = 0$

2. **return** $\text{COPY1}((\mathsf{dg}, \mathsf{idx}))$

$\text{COPY1}_{\mathsf{idx}}(\mathsf{ref_in})$

1. $(\mathsf{dg}, \mathsf{idx}) \leftarrow \text{DEREFERENCE}_{\mathsf{idx}}(\mathsf{ref_in})$

2. **if** $\mathsf{sba}[\mathsf{idx}].\mathsf{copy} \neq \mathit{nil}$ **then**

  2.1. **return** $\mathsf{sba}[\mathsf{idx}].\mathsf{copy}$

3. $\mathsf{new_idx} \leftarrow \mathsf{free_idx}$; $\mathsf{free_idx} \leftarrow \mathsf{free_idx} + 1$

4. $\mathsf{newcopy} \leftarrow$ **new** *Node*

5. $\mathsf{newcopy.type} \leftarrow \mathsf{sba}[\mathsf{idx}].\mathsf{type}$

6. $\mathsf{sba}[\mathsf{idx}].\mathsf{copy} \leftarrow (\mathsf{newcopy}, \mathsf{new_idx})$

7. $\mathsf{arcs} \leftarrow \{\text{ABSARC}(a, \mathsf{idx}) \mid a \in \mathsf{dg.arcs}\} \cup \mathsf{sba}[\mathsf{idx}].\mathsf{comp_arcs}$

8. **for each** $(\mathsf{label}, \mathsf{ref})$ **in** arcs **do**

  8.1. $(\mathsf{dg1}, \mathsf{ch_idx}) \leftarrow \text{COPY1}_{\mathsf{idx}}(\mathsf{ref})$

  8.2. *Push* $(\mathsf{label}, (\mathsf{dg1}, \mathsf{ch_idx} - \mathsf{new_idx}))$ *into* newcopy.arcs

9. **return** $(\mathsf{newcopy}, \mathsf{new_idx})$

Figure 6.6 Copy algorithm

however, we can eliminate the need to maintain multiple copies of such a graph by simply assigning each instance a new block of scratch buffers. This property also simplifies the implementation of structure sharing, which we will discuss next.

6.3.2 Structure Sharing

The changes that are required to make the indexing algorithm capable of structure sharing are analogous to the variant of Tomabechi's algorithm presented by Malouf et al. (2000) discussed in Section 2.2.3. The conditions under which structure sharing is possible are slightly different, though. As we explained before, a node in a graph is uniquely identified by its index. This property allows us to share grammar nodes without any of the problems mentioned in Section 2.2.3. Nodes from different graphs will always be assigned different scratch buffers. But even a single grammar node can appear as different copies within one graph. After unifying multiple instances of the same graph, the same node may end up representing multiple instances of this node in the result graph. As long as the different instances acted as different nodes during unification, they will also obtain different indexes in the result graph. Since multiple instances of a same graph are always assigned different blocks of indexes[3] we do not need any additional logic to allow this kind of sharing.

A drawback of the indexing technique is that we need to introduce a different condition for structure sharing instead. The offset data that is included in the graph is a part of the

[3]For example, MAKEWELLFORMED allocates new scratch buffers for each constraint each time it interleaves a unification.

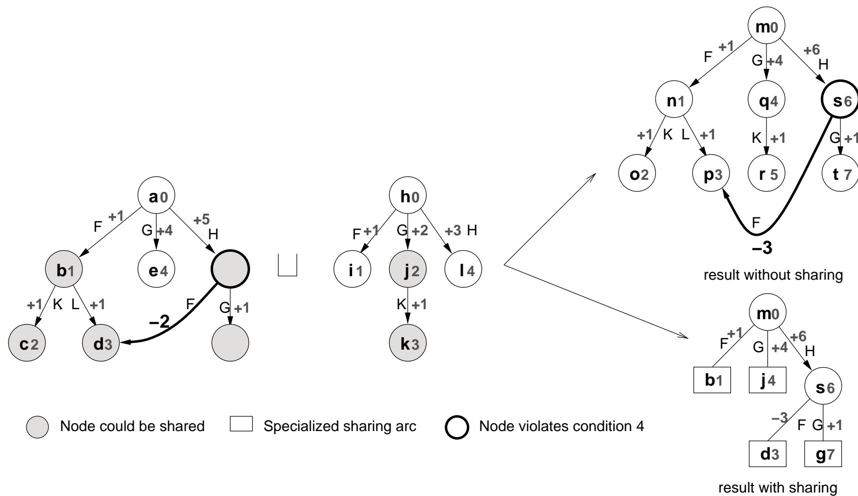


Figure 6.7 Possible violation of condition 4. Node **f** cannot be shared, as this would cause the arc labeled F to derive an index colliding with node **q**.

definition of the graph. This means that we can only share an input subgraph if this data is identical to the result graph as well. Because we assumed the offset data was included as relative offsets in the arcs, the condition is as follows:

4. all arcs in the shared subgraph must have the same offsets as the subgraph that would have resulted from copying.

The first three conditions are identical to the first three conditions for Malouf et al.'s algorithm (see Page 27).

A possible violation of this constraint is shown in Figure 6.7. This condition can typically only be violated in case of reentrancy. Basically, the condition can be violated when a reentrancy points past a node that is bound to a larger subgraph.

Note that if, in the example of Figure 6.7, there was no arc from node **f** to node **d**, the sharing results would be the same for the original and the indexing algorithm. Because new nodes are introduced in the subgraph of **a**, node **a** cannot be shared (condition 3) with either algorithm. Apart from that, none of the offsets associated with the arcs of respectively graph **b**, **j**, and **f** would change, so condition 4 would not be violated.

Just as with Malouf et al.'s approach, the complexities of structure sharing are hidden in the COPY1$_{\text{idx_sh}}$ function. A version of COPY1$_{\text{idx_sh}}$ that implements structure sharing for the indexing algorithm is shown in Figure 6.8. COPY$_{\text{idx_sh}}$ is very similar to COPY$_{\text{idx}}$. The only difference is the inclusion of the checks for the four conditions. The conditions 1 through 3 are checked at line 7 and 9.3. Condition 4 is checked at line 9.4 by comparing the offsets of the old arc and new arc.[4]

[4]Note that for reasons of convenience we compute the offset of the source arc (idx1 − idx) rather than using

$\text{COPY1}_{\text{idx_sh}}(\text{ref_in})$

1. $(\text{dg}, \text{idx}) \leftarrow \text{DEREFERENCE}_{\text{idx_sh}}(\text{ref_in})$
2. **if** sba[idx].copy $\neq$ *nil* **then**
  2.1. **return** sba[idx].copy
3. new_idx $\leftarrow$ free_idx; free_idx $\leftarrow$ free_idx $+ 1$
4. newcopy $\leftarrow$ **new** *Node*
5. newcopy.type $\leftarrow$ sba[idx].newType
6. sba[idx].copy $\leftarrow$ (newcopy, new_idx)
7. share $\leftarrow$ sba[idx].comp_arcs $= \emptyset \wedge$ dg.type $=$ sba[idx].newType
8. arcs $\leftarrow \{\text{ABSARC}(a, \text{idx}) \mid a \in \text{dg.arcs}\} \cup$ sba[idx].comp_arcs
9. **for each** (label, (dg1, idx1)) **in** arcs **do**
  9.1. $(\text{ch_dg}, \text{ch_idx}) \leftarrow \text{COPY1}_{\text{idx_sh}}((\text{dg1}, \text{idx1}))$
  9.2. *Push* (label, (ch_dg, ch_idx $-$ new_idx)) *into* newcopy.arcs
  9.3. **if** dg1 $\neq$ ch_dag **then**
    share $\leftarrow$ *false*
  9.4. **if** idx1 $-$ idx $\neq$ ch_idx $-$ new_idx **then**
    share $\leftarrow$ *false*
10. **if** share **then return** (dg, new_idx)
  else return (newcopy, new_idx)

Figure 6.8 Copy algorithm with structure sharing.

6.3.3 Other Optimizations

Since Deltra is a DCG, it uses fixed arity unification. This means that each nodes has a fixed set of features that can be laid out linearly in memory, not requiring arc structures. This means that the reduction in memory usage can be even greater than with LinGO. In addition, we introduced some additional optimizations the indexing technique for Deltra, which we will present next.

Deferred Copying Just as we use an array of scratch buffers for unification and copying, we can also use an array of scratch buffers for equality checking. Tomabechi's algorithm requires that the graph resulting from unification be copied before it can be used for further processing. This can result in superfluous copying when the graph, and the corresponding item, is already contained on the chart. Our technique allows equality checking to use the bindings generated by $\text{UNIFY1}_{\text{idx}}$ in addition to its own buffers. This allows us to defer copying until we completed subsumption checking.

Compressed Storage Rrepresentation of Nodes With a straightforward implementation of our algorithm for Deltra, we obtain a node size of 8 bytes.[5] By dropping the concept of a fixed node size, we can reduce the size of atom and bottom nodes to 4 bytes.

the stored value.

[5]Deltra does not have a type hierarchy.

Type information can be stored in two bits. We use the two least significant bits of pointers (which otherwise are 0) to store this type information. Instead of using a pointer for the value field, we store nodes in place. Only for reentrancies do we still need pointers. Complex nodes require 8 bytes, as they include a pointer to the first node past its children (necessary for unification). This scheme requires some extra logic to decode nodes, but significantly reduces memory consumption.

6.4 Performance Comparison

In this section we will analyze the performance of the proposed algorithms. We will focus in particular on the memory consumption and execution times. We show that our parser can compete with other parsers for LinGO by comparing it to an implementation of CHEAP (Callmeier, 2000).

6.4.1 Results for LinGO

The performance results were obtained using CaLi in combination with three different versions of the unification algorithm: traditional Tomabechi with Malouf et al.'s structure sharing approach, a hashed version of Tomabechi, and an algorithm implementing the indexing technique. Tests were performed for the 'csli', 'aged', as well as the 'fuse' test suites. All tests were run on a Red Hat Linux box with a dual 500MHz Pentium-III Katmai processor with 512MB of main memory.

The implementations for both the indexing and hashing algorithms use 4 byte nodes and 8 byte arcs. Each non-leaf node is associated with an additional 4 bytes for a pointer to an arc list. The offset fields required for the indexing algorithm were included in two pad bytes of the arc structure. The implementation of Tomabechi's algorithm also has an arc size of 8, but uses a 24 byte node structure. All implementations use structure sharing.

Memory consumption was measured by counting the maximum number of nodes and arcs simultaneously stored during a single parse, and multiplying these figures by the respective structure sizes. This measure indicates the minimum size of the buffer required to store the arcs and nodes for a single parse. We did not include the statically allocated buffers to store, for example, the grammar and the scratch buffers. The size of the scratch buffers that are needed for both the indexing and the hashing technique is negligible. In the case of the indexing version, the total memory consumed by the buffers is less than 32K. The hashing version requires slightly more memory, because it has an additional lookup buffer that needs to have sufficient entries for hashing to be efficient. In both cases, though, the reduced size of the grammar more than compensates for the extra memory needed for these buffers.

Table 6.1 shows the results of a comparison between a plain version of Tomabechi and one with hashing. On average, the hashing technique reduces the memory requirements by over 50%. Execution times, on the other hand, are increased. The increase in execution time is the highest for longer sentences. In these cases, the chance that more nodes will map to the same bucket increases.

Suite	Tomabechi tasks ϕ	Tomabechi time ϕ (s)	Tomabechi space ϕ (kb)	Hashed time ϕ (s)	Hashed space ϕ (kb)	reduction time %	reduction space %
aged	**39463**	**0.14**	**2391**	**0.16**	**1170**	**-8.5**	**51.1**
csli	**9491**	**0.03**	**644**	**0.04**	**312**	**-7.9**	**51.6**
fuse	**160924**	**0.56**	**8849**	**0.61**	**4397**	**-10.3**	**50.3**

(generated by [incr tsdb()] at 13-aug-2000 (21:24 h))

Table 6.1 Performance results for hashing technique relative to an implementation of Tomabechi's algorithm.

Suite	Tomabechi tasks ϕ	Tomabechi time ϕ (s)	Tomabechi space ϕ (kb)	Indexed time ϕ (s)	Indexed space ϕ (kb)	reduction time %	reduction space %
aged	**39463**	**0.14**	**2391**	**0.14**	**1203**	**-1.3**	**49.7**
csli	**9491**	**0.03**	**644**	**0.04**	**303**	**-1.7**	**53.0**
fuse	**160924**	**0.56**	**8849**	**0.57**	**4859**	**-1.7**	**45.1**

(generated by [incr tsdb()] at 13-aug-2000 (21:24 h))

Table 6.2 Performance results for indexing technique relative to an implementation of Tomabechi's algorithm.

As can be seen in Table 6.2, using the indexing technique has less impact on the execution times. Moreover, the impact on the execution times is independent of the size of the sentence. The reduction in memory usage that can be obtained with sharing of grammar nodes gets less for sentences that require more tasks to complete. This explains the different results for the different test suites.

In order to verify the overall performance of our parser, we compared our implementation of Tomabechi to a version of CHEAP. The latter version also uses the **KD** parsing schema, uses the same filtering techniques, and uses Tomabechi's algorithm for unification. Altogether, CHEAP provides a good gold standard for the performance evaluation of our implementation. Table 6.3 shows a comparison between the performance of CaLi and

Suite	CHEAP time ϕ (s)	CHEAP space ϕ (kb)	CaLi time ϕ (s)	CaLi space ϕ (kb)	reduction time %	reduction space %
aged	**0.17**	**2294**	**0.14**	**2391**	**13.5**	**-4.2**
fuse	**0.62**	**7989**	**0.56**	**8849**	**10.8**	**-10.8**

(generated by [incr tsdb()] at 14-aug-2000 (17:50 h))

Table 6.3 Comparison of performance between the sequential version of CaLi and CHEAP.

CHEAP. Differences in memory utilization between our Tomabechi-based implementation and CHEAP can be attributed to the fact that CHEAP and CaLi use a different representation for arcs. Execution times are comparable, considering we are dealing with two different implementations.

6.4.2 Results for Deltra

The experiments for the Deltra variant were run on a Pentium III 600EB (256 KB L2 cache) box, with 128 MB memory, running Linux. We used a test set of 22 sentences of varying length. Usually, approximately 90% of the unifications failed.

We tested both memory usage and execution time for various configurations. The results are shown in Figure 6.9. It includes a version of Tomabechi's algorithm. The node size for this implementation is 20 bytes. For the proposed algorithm we have included several versions: a basic implementation, a version with compressed nodes, a version with deferred copying, and a version with structure sharing. The basic implementation has a node size of 8 bytes, the others have a variable node size. Whenever applicable, we applied the same optimizations to all algorithms. We also tested the speedup on a dual Pentium II 266 MHz.[6] Each processor was assigned its own scratch tables. Apart from that, no changes to the algorithm were required.

The memory utilization results show significant improvements for our approach.[7] Packing decreased memory utilization by almost 40%. Structure sharing roughly halved this once more.[8] The fourth condition prohibited sharing in less than 2% of the cases where it would be possible in Tomabechi's approach.

Figure 6.9 shows that our algorithm does not increase execution times. Our algorithm even scrapes off roughly 7% of the total parsing time. This speedup can be attributed to improved cache utilization. We verified this by running the same tests with cache disabled. This made our algorithm actually run slower than Tomabechi's algorithm. Deferred copying did not improve performance. The additional overhead of dereferencing during equality checking was not compensated by the savings on copying. Structure sharing did not significantly alter the performance as well. Although this version uses less memory, it has to perform additional work. Details of the implementation for Deltra can be found in (van Lohuizen, 2000).

6.5 Comparison to other Algorithms

We reduced memory consumption of popular graph unification algorithms, as presented by Tomabechi (1991) and Wroblewski (1987), by separating scratch fields from node structures. Pereira's (1985) algorithm also stores changes to nodes separately from the graph.

[6] These results are scaled to reflect the speedup relative to the tests run on the other machine.

[7] The results do not include the space consumed by the scratch tables. However, these tables do not consume more than 10 KB in total, and hence have no significant impact on the results.

[8] Because the version with compressed nodes has a variable node size, structure sharing yielded less relative improvements than when applied to the basic version. In terms of number of nodes, though, the two results were identical.

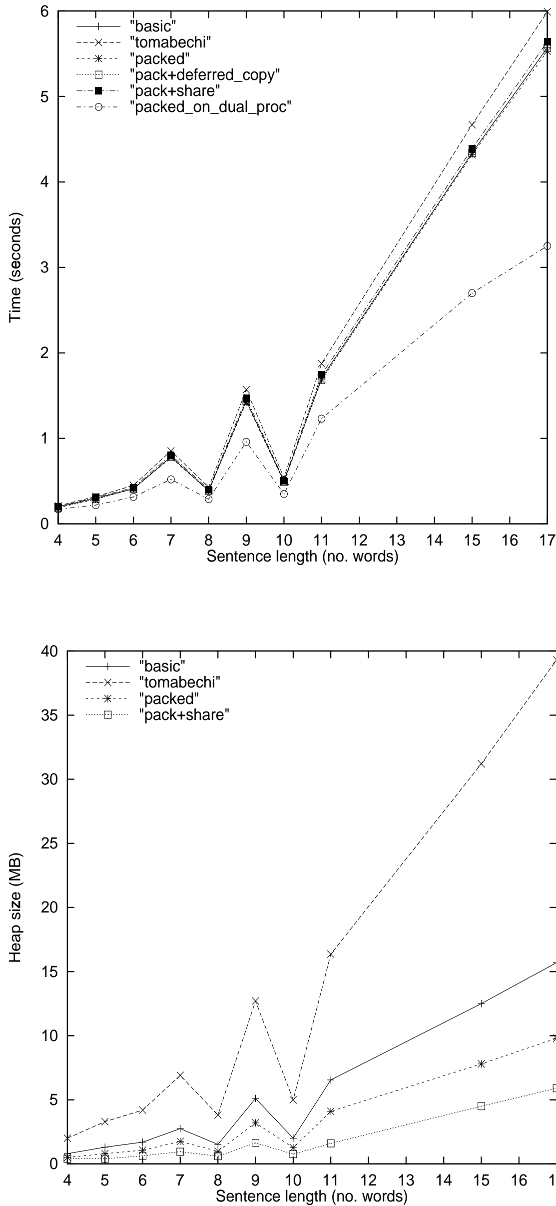


Figure 6.9 Results for Deltra. Top: execution time in seconds. Bottom: memory usage of the graph heap in MB.

However, Pereira's mechanism incurs a $\log(n)$ overhead for accessing the changes (where n is the number of nodes in a graph), resulting in an $O(n \log n)$ time algorithm. The indexing algorithm runs in $O(n)$ time.

With respect to over and early copying (as defined in (Tomabechi, 1991)), our algorithm has the same characteristics as Tomabechi's algorithm. In addition, our algorithm allows the copying of graphs to be postponed until after the verification step is completed. This would require additional fields in the node structure for Tomabechi's algorithm.

Our algorithm allows sharing of grammar nodes, which is usually impossible in other implementations (Malouf et al., 2000). A weak point of our structure sharing scheme is its extra condition. For the Deltra grammar, this proved to be no limitation, as structure sharing could be exploited as usual in most cases. For LinGO, the percentage of failed shares due to condition 4 was considerably higher, although the ability to share grammar nodes roughly compensated for this.

Finally, contrary to most other algorithms for unification, both the hashing and the indexing algorithms are thread-safe. This makes them suitable for concurrent unification. As the presented algorithms are based on Tomabechi's algorithm, they are also suitable for parallel unification (cf. (Fujioka et al., 1990)).

6.6 Conclusions and Future Directions

Memory consumption is a major concern with many of the current unification-based grammar parsers. The presented algorithm provides a fast and memory-efficient alternative to Tomabechi's algorithm. We have shown how memory usage can be reduced by separating scratch fields from nodes.

A fast alternative to Tomabechi is the indexing technique. Although this technique requires extra computation, it still obtains roughly equivalent execution times. We showed that improved cache utilization can work to the advantage of the indexing technique. As current developments indicate that the difference between cache and memory speed will only get larger, this effect is not just an artifact of current architectures.

With the hashing technique we obtained a slightly higher reduction in memory utilization than with the indexing technique. However, measurements indicate that there is roughly a 25% possible reduction in memory utilization—using the indexing technique—if the failures of sharing that are caused solely by condition 4 can be eliminated. A possible technique to accomplish this is to reserve index numbers at critical positions in a graph. These can then be assigned to nodes in later unifications without introducing conflicts elsewhere. For example, the failure to share node **f** in Figure 6.7 could have been prevented by allocating unused indexes at node **e**. The type hierarchy can be a useful source of information to determine for which nodes to reserve indexes.

We showed that—without increasing execution times—an additional reduction in memory utilization can be obtained by compressing the node and arc structures. Similar techniques could be applied to the indexing algorithm for LinGO. A small reduction in memory can be achieved by storing atom nodes in arc structures. The value field in arc is implemented

as a pointer to a node. Since a node structure is only 4 bytes, we could include it in this field. This would save 4 bytes for each atom node. Another approach to reduce memory utilization would be to reduce the arc size. As there are more arcs than nodes, and given that the arc structure is twice the size of a node structure, arcs are responsible for the majority of the memory consumption. The arc size could be reduced, for example, by eliminating the label field, deriving the required information from the node's type instead.

Chapter 7

Design and Implementation of a Parallel Parser

In this chapter, we present the design and implementation of a parallel parser that is able to achieve considerable speedup. The design is based on the analysis presented in Chapter 4 and Chapter 5. The implementation uses the thread-safe algorithm presented in the previous chapter to enable concurrent unification. An abridged version of this chapter has been published in (van Lohuizen, 2001b). An approach to parallel parsing specifically for Deltra can be found in (van Lohuizen, 1999).

7.1 Introduction

As was discussed in Section 2.3.3, there are many possible approaches to parallel chart parsing. The results of the analyses presented in Chapter 4 and 5 give an indication of which approaches to parallel chart parsing can be fruitful. In Chapter 4, it was shown that—both for the LinGO and Deltra grammars—to ensure sufficient parallelism is available, parallelism should be exploited at the fine granularity of single unification operations. In Chapter 5, we saw that, for the LinGO grammar, using a greedy grouping heuristic as a means to distribute work amongst processors can greatly reduce the communication between processors. We also mentioned that for the Deltra grammar, the same effect can be obtained by using the rule-based heuristic.[1] Both the greedy and rule-based distribution technique require the use of a distributed chart. As was mentioned in Section 2.3.3, a distributed chart as opposed to a centralized chart can have the additional advantage of reducing synchronization overhead.

Besides focusing on the ability to achieve speedup, we need to consider the following. Over the past years, improvements to existing parsing techniques have boosted the performance of parsers by orders of magnitude (Oepen and Callmeier, 2000). It is very well possible that this development will continue for some time to come. Tying the design of a parallel parser too much to a particular approach to parsing may make it hard to incorporate such improvements as they become available. To allow for such improvements without significantly changing the design of the parallel parser, a solution to parallel parsing should be as generic as possible. The design of the parallel parser that is presented in this chapter is greatly influenced by this design criterion.

In summary, the design of the parser presented in this chapter is based on three main design principles:

[1]For Deltra, the rule-based grouping heuristic has the additional advantage that it will group all items that can be involved in a single verification step.

1. Work should be distributed at the granularity of single unification operations.

2. The design should be based on a distributed chart to allow the greedy and rule-based grouping heuristics to be used in combination with, respectively, the LinGO and Deltra grammar.

3. The approach to parallel parsing should not be tied to a particular parsing schema or grammar formalism, and should allow optimizations for sequential parsers to be incorporated as much as possible.

In the following sections, we will present an approach to parallel parsing for both the Deltra and LinGO grammars. As with the previous chapters, the focus will be on a solution for the LinGO grammar. The difference with respect to the Deltra grammar will be explained as appropriate. As suggested in Chapter 4, the parser will allow distribution of work at the granularity of a single unification operation, corresponding to a type 1 task graph. Following the suggestions of Chapter 5, the LinGO and Deltra parser will utilize the greedy and rule-based grouping heuristics, respectively. To allow for concurrent unification, both parsers will utilize the indexing technique for unification presented in the previous chapter. For Deltra, we used compressed nodes, as presented in Section 6.3.3. Section 7.2 and 7.3 discuss respectively the design and implementation of the parallel chart parser. In Section 7.4, a theoretical analysis of the presented scheduling technique is given. Finally, Section 7.5 presents performance results based on the implementation of the system.

7.2 Scheduling

An obvious approach for increasing the likelihood that optimizations for sequential parsers can be used in a parallel parser as well, is to let a parallel parser mimic a sequential parser as much as possible. This can be accomplished by letting each processor run a conventional sequential parser augmented with a mechanism to combine the individual results. In such a scenario, each processor, or thread,[2] has its own agenda and chart. Taking this approach therefore automatically satisfies the second requirement.

Let us first take a closer look at the proposed approach, assuming we use the LinGO grammar. Initially, each thread is assigned a different set of work, for example, representing a different part of the input string. Each thread first proceeds as it would in the sequential case, that is, executing the unify–verify–match cycle until all entries on the agenda have been processed. After completing the work on its agenda, a thread will enter the communication phase and match the edges on its chart with the edges derived so far by the other threads. This may produce new unification tasks, which the thread puts on its agenda. After the communication phase is completed, it returns to normal parsing mode to execute the work on its agenda. This process continues until all edges of all threads have been matched against each other and all work has been completed.

[2]As we will assign a single thread to each processor, using the threading library of the host operating system, we will use the two terms interchangeably.

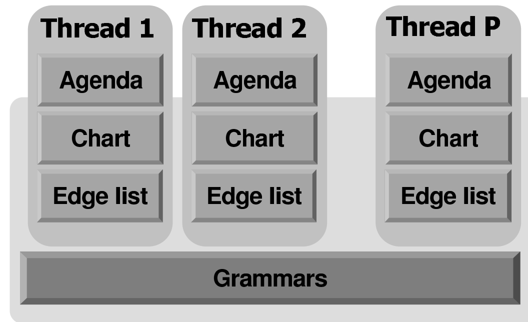


Figure 7.1 Distribution of data structures in the MACAMBA system.

The presented scheme implicitly implements a variant of the greedy grouping heuristic presented in Section 5.3.2. We may expect that this variant is more capable of reducing the maximum total communication than the greedy heuristic presented in Chapter 5. Just as with the greedy heuristic, a unification task is always executed on a processor that owns at least one of the input items. However, contrary to the specification of the greedy heuristic, the result of a unification task will *always* be stored on the processor that computed the unification. Despite these additional restrictions, we may still expect this approach to yield a well-balanced distribution. The additional restrictions are compensated by allowing the dynamic assignment of unification tasks to one of the corresponding input items, rather than fixing them at random.[3]

For the Deltra grammar, we use the rule-based grouping heuristic. In this case, rather than adding each newly derived item to its own chart, a thread will send it to the thread owning the group associated with the item. This is the only change required to implement the rule-based grouping heuristic. The changes required to implement this scheme, relative to an implementation for LinGO, are minimal.

So far, we have completely ignored the first requirement. In Section 3.2.2, we showed that work stealing can be a viable technique to accomplish a well balanced distribution of work with a minimum of overhead. In the presented design, the fine-grained distribution of single unification tasks can be enabled by allowing idle threads to steal work from other threads that still have work in their work queues.

In the next sections, we will explain the details of the presented scheduling technique. First we explain how the integrity of the data structures is enforced. We then focus on the scheduling algorithm. We also discuss work stealing in more detail.

7.2.1 Data Structures

Figure 7.1 shows an outline of the data structures involved in our approach and how they are distributed amongst the threads. Each thread has its own chart and agenda. In addition, each thread has an **edge list**, which is a linked lists of the items it derived. This list is used to communicate the derived items to other processors. Such a list is often already present in the chart structure. Nevertheless, since it is an integral part of the communication mechanism, we mention it separately.

Each thread has read and write access to its own agenda, chart, and edge list, and has read-only access to the respective structures of all other threads. Grammars are read-only and can be read by all threads.[4]

In the communication phase, threads use the edge lists of other threads to figure out which edges have been derived by those threads. These **foreign edges** are matched against the edges on the local chart (**local edges**). All this can be done with read-only access to the edge list structures. However, the next step can pose a problem. When a match succeeds, a thread will put a task on the agenda corresponding to the unification of the feature structures of to the local and foreign edges. Many unification algorithms make use of scratch fields in the graph structures. Such algorithms can therefore not be used for concurrent unification. To circumvent the problem, we use the thread-safe unification algorithm based on the indexing technique presented in Section 6.3.

The distributed agendas that are used in this approach may make it hard to implement the strict control over the order of evaluation of tasks that is required by some parsers. One solution to the problem would be to use a centralized agenda. The disadvantage of such a solution is that it might increase the synchronization overhead. In Section 3.2.2, we presented some scheduling algorithms that can mitigate the overhead in such a scenario.

7.2.2 Scheduling Algorithm

At startup, each thread calls the scheduling algorithm shown in Figure 7.2. This algorithm can be seen as a wrapper around an existing sequential parser that takes care of combining the results of the individual threads. The functionality of the sequential parser is embedded in Step 1.2. After this step, the agenda will be empty. The communication between threads takes place in Step 1.3. Each time a thread executes this step, it will inspect all the newly derived foreign edges of other threads and match them with the edges on its own chart. To avoid revisiting previously processed items, each thread records the last visited edge of the edge list of each other thread. As a result of Step 1.3, the agenda may become non-empty. In this case, newWork will be set and Step 1.2 is executed again. This cycle continues until all work is completed.

The remaining steps serve several purposes: preventing double work, load balancing, and detecting termination. We will explain each of these aspects in the following sections.

[3] It is not possible to define the presented distribution scheme in terms of a grouping heuristic, as the definition of grouping heuristics does not allow for dynamic restrictions. The greedy grouping heuristic can be seen as an approximation of this scheme.

[4] MACAMBA allows for multiple grammars to be used simultaneously. This feature is used by Deltra.

```
global shared Generation ← P + 1, Terminate ← false

SCHED()
  var threadGen ← threadNumber, token
  var newWork ← true, communicated ← false, victimized ← false
  if threadNumber = 1 then token ← 0
                       else token ← nil
  while not Terminate do
      1. while newWork do
            1.1. newWork ←false
            1.2. Process the agenda as in the sequential case. In addition, for each newly
                 derived edge I, stamp I by setting I.generation to the current value for threadGen,
                 add I to this thread's edge list, and set communicated to true.
            1.3. Examine all other threads for newly derived edges. For each new edge I and
                 for each edge J on the chart for which holds I.generation > J.generation, add the
                 corresponding task to the agenda if it passes the filter. If any edge was processed,
                 set newWork to true.
            1.4. lock GlobalLock
            1.5. threadGen ← Generation ← Generation + 1
            1.6. unlock GlobalLock
      2. newWork ←STEAL()
      3. if not newWork and token ≠ nil then
          3.1. if communicated or victimized then
                 Pass token with value 0 to next in ring and set token to nil.
                 communicated ← victimized ← false
          3.2. elseif token = P then
                 Terminate ← true
          3.3. else
                 Pass token with value token + 1 to next in ring and set token to nil.
      4. if New items added to foreign edge lists then
          newWork ← true
```

Figure 7.2 Scheduling algorithm.

7.2.3 Preventing Duplicate Matches

When two matching edges are stored on the charts of two different threads, the scheduling algorithm should prevent both threads performing the corresponding match. Failing to do so can cause the derivation of duplicate edges and eventually a combinatorial explosion of work. Our solution is based on a generation scheme. Each newly derived edge is stamped with the current generation of the respective thread, threadGen (see Step 1.2). A thread will only perform the match for two edges if the edge on its chart has a lower generation than the foreign edge (see Step 1.3). Obviously, because the value of threadGen is unique for each thread (see Step 1.5), this scheme prevents two edges from being matched twice.

SCHED also ensures that two matching edges will always be matched by at least one thread. After a thread completes Step 1.3, it will always raise its generation. The new generation will be greater than that of any foreign edge processed before. This ensures that when an edge is put on the chart, no foreign edge with a higher generation has been matched against the respective chart before.

Theorem 7.1 *The scheduling algorithm* SCHED *ensures that any two items I and J, where $I \otimes J$, will be matched by exactly one thread.*

Proof. We first prove that SCHED matches any two items I and J on at most at one processor. This follows from the fact that a generation is unique for one processor. By Step 1.5, if I and J are stored on different processors, they are guaranteed to have different generations. By Step 1.3, the match cannot be performed by both the thread that derived I and the thread that derived J. Given that the items in the edge lists are only matched against the chart once (Step 1.3), I and J will also never be matched more than once.

SCHED also ensures that at least one thread processes the match for I and J. Without loss of generality, assume that I was derived by thread T_a while having a generation of g_a and that J was derived by thread T_b while having a generation of g_b, where $g_a < g_b$. Because $g_a < g_b$, by Step 1.3, only T_a can perform the match of I and J. We need to consider two cases: one where T_a adds I to its agenda *before* it finds and processes J on T_b's edge list, and one where T_a adds I to its agenda *after* it finds and processes J on T_b's edge list. In the former case, T_a will obviously find I on its chart as it processes J. The latter case, however, cannot occur. After processing J, T_a will always execute Step 1.5. This causes the generation of T_a to become larger than g_b, the generation associated with J. As the generation associated with T_a can never decrease, I is guaranteed to have a higher generation than J. □

In practice, we only need to increase threadGen when a thread is about to add an item to its chart and when the highest generation encountered with any foreign edge processed so far by the respective thread is higher than the current value of threadGen. This allows for a more lazy update of the generation counter and allows the maximum number of times GlocalLock is locked to be bound by the number of items put on the chart.

This scheme will work for any parsing schema where at most two edges are involved in the deduction of a new edge. This means this scheme will work for the majority of popular parsing schemata and, hence, complies with our requirement of providing a solution not tied to a particular parsing schema or grammar.

STEALFROMFOREIGNEDGELISTS(victim)

1. **if** *locking of* EdgeListsLock *fails* **then**
   return *false*
2. *Attempt to steal edges foreign to* victim, *preferably from own edge list, for matching with the victim's chart and set* victim.victimized *to true if successful.*
3. **unlock** EdgeListsLock
4. **if** *no work was stolen* **then**
   return *false*
5. **lock** ChartLock
6. *Set the minimum generation for all successive items the victim will add to the chart to the maximum generation of any of the stolen foreign items.*
7. **unlock** ChartLock
8. *Execute the stolen matches and put any resulting tasks on own agenda.*
9. **return** *true*

STEALFROMAGENDA(victim)

1. **if** *locking of* AgendaLock *fails* **then**
   return *false*
2. *Attempt to steal part of the work on the agenda of* victim *and set* victim.victimized *to true if successful.*
3. **unlock** AgendaLock
4. **if** *no work was stolen* **then**
   4.1. **return** *false*
5. *Add all but one stolen task to thief's agenda.*
6. *Execute the stolen task not put on the agenda.*
7. **return** *true*

Figure 7.3 Stealing algorithms

7.2.4 Work Stealing

In the detailed design as presented so far, each thread exclusively executes the unification tasks on its agenda. Obviously, this violates the requirement that each unification task be scheduled dynamically. In Section 3.2.2, we presented work stealing as an effective mechanism to distribute work amongst processors at a fine granularity. With this technique, a thread will first attempt to steal work from the queue of another thread before announcing itself to be idle. If it succeeds, it will resume normal execution as if the stolen tasks were its own.

A thread becomes a thief by calling STEAL, at Step 2 of SCHED. STEAL will first randomly select a thread, which we refer to as the victim. It then attempts to steal any work from the victim such that the following rules are obeyed.

1. If some work is stolen, the thief should execute at least one task of the stolen work itself.

2. Before the work should be visibly removed from the victim, the victimized variable of the victim should be set to *true*.

The first rule ensures that no livelock can occur, that is, it prevents some task being stolen repeatedly, without ever being executed. The second rule is necessary to allow the termination algorithm to detect the transfer of work properly. Termination is discussed in more detail in Section 7.2.5.

STEAL will attempt to steal work from two different sources: the agendas, which contain outstanding unification tasks, and the unchecked foreign edges, which resemble outstanding match tasks between threads. Figure 7.3 shows the stealing algorithms for both cases. With STEALFROMFOREIGNEDGELISTS, an attempt is made to steal any of the outstanding match tasks. A thread will first attempt to steal items originating from its own edge list. The idea is that this can increase cache performance, as the thief might still have the respective items in its cache. With STEALFROMAGENDA, a thief will steal *half* of the work on the agenda. This balances the load between the two threads and minimizes the chance that either thread will have to call the expensive steal operation soon thereafter.

Note that when match tasks are stolen and performed by another thread than the owner, the prevention of double work mechanism presented in the previous section will break. To solve this problem, a thief tells the victim the highest generation of any foreign item it has stolen. Each thread should lock each addition of an item to the chart and raise its generation within the lock if necessary. This ensures that by the time thread T adds an item I to the chart, all foreign items of T that were ever processed had a smaller generation than I. This mechanism can be implemented without much additional overhead if the optimization of the prevention of double work scheme, proposed in the previous section, is implemented. This optimization already required the generation to be checked at the time of adding an item to the chart.

With Deltra, we allow an additional mode of stealing. The Deltra parser reduces the number of tasks on the agenda by putting tasks on the agenda just before the match phase of the unify–verify–match cycle. An entry on the agenda is therefore defined as a set of unification operations. A thread that processes such a task loops over all operations in the set. In order to allow stealing of single operations, threads should be allowed to steal single iterations of this loop. To this extent, a signature is set at the start of each loop to tell potential thieves how to obtain an iteration of the loop.

The stealing functions for the different sources are incorporated into STEAL as follows. After selecting the victim, STEAL will first attempt to steal some of the victim's match tasks. If it succeeds, it will perform the matches and put any resulting unification tasks on its own agenda. Otherwise, it will attempt to steal work from the victim's agenda. If this does not succeed, it will return failure. In case of Deltra, it will also attempt to steal single iterations. Note that with SCHED, idle threads will keep calling STEAL until they either obtain new work or all other threads become idle.

Obviously, stealing eliminates the exclusive ownership of the agenda and unchecked foreign edge lists of the respective threads. As a consequence, a thread needs to lock its agenda and edge lists each time it needs to access it. In addition, for the Deltra parser, each iteration necessary to processes an agenda entry requires additional locking. According to the work-first principle (see Section 3.2.3), we can improve performance by moving as much of the overhead from victim to thief as possible. Therefore, synchronization between worker and thief can be optimized by using a Dijkstra like asynchronous mutual exclusion protocol (Dijkstra, 1965, Frigo et al., 1998). For completeness, the protocol we use is illustrated in figure 7.4. As long as no stealing is taking place, workers will not have to resort to an expensive lock. Most of the locking is done by workers or victims; only a small fraction is done by thieves. Thus we reduce the total amount of work by moving some of the overhead

```
      Worker/Victim                    Thief

1  isBusy=YES;                 10 Determine victim
2  if (L == YES)               11 victim.lock;
3     isBusy=NO;               12 victim->L = YES;
4     self.lock;               13 while (!victim->isBusy) {}
5     Protected block          14 Protected block
6     self.unlock;             15 victim->L = NO;
7  else                        16 victim.unlock;
8     Protected block
9     isBusy=NO;
```

Figure 7.4 Pseudocode of Victim/Thief protocol. `L` is set by the thief to indicate a lock request. A memory fence is needed between line 1 and 2 to ensure that both parties see changes to main memory in the right order.

on to the critical path.

7.2.5 Termination

It is crucial that no idle thread terminate while any other thread is still processing tasks. The processing of a single task can result in considerable amounts of new work. The premature termination of a thread can therefore lead to a waste of computing resources. A thread may terminate when all work is completed, that is, if and only if all of the following conditions hold simultaneously:

- all agendas of all threads are empty,
- all possible matches between edges have been processed, and
- no thread is executing Step 1.2 or 1.3.

To detect termination we use a token passing scheme similar to the scheme proposed by Dijkstra et al. (1983). Such a scheme ensures both that all threads terminate when all work is completed and that no thread terminates when there is still work being done at any thread. With Dijkstra's algorithm, termination is detected by a designated thread. We allow any thread to detect termination. In addition to the communication by message sending in Dijkstra's protocol we also allow work stealing. To enable the victims to keep track of the fact that work was stolen, thieves must set the victimized variable.

Theorem 7.2 SCHED *terminates for any parsing system that yields a finite amount of items.*

Proof. Given there is a finite amount of work, a thread is guaranteed to leave the loop at Step 1 at some point. By requiring that a thread calling STEAL will process at least one of the tasks it steals itself, we ensure that there can be no livelock between threads stealing

work. What remains to be proven is that Terminate will eventually be set to *true* so that threads will leave the outer loop.

Assume all threads have completed all work and have left the loop at Step 1. Because, under these conditions, no thread will add new items to its edge list and because it is not possible to steal work from any thread, newWork is guaranteed to remain *false*. For each thread, the value of communicated ∨ victimized can be either *true* or *false*. Since there is no more work being processed, it is guaranteed that communicated and victimized can never be set from *false* to *true*. Since a thread will set communicated and victimized to *false* at Step 3.1 when it gets the token, it is guaranteed that after at most P passings of the token communicated ∨ victimized is *false* for all threads. In this state, each successive passing of the token will result in raising the token value at Step 3.3. This continues until the token is P, in which case the thread owning the token will set Terminate to *true* at Step 3.2. □

The proof that Terminate will not be set to *true* before all work is finished is analogous to the proof presented by Dijkstra (1983).

7.3 Implementation

The implementation of the system consists of three parts: MACAMBA, CaLi, and Deltra. MACAMBA stands for Multi-threading Architecture for Chart And Memoization-Based Applications. The MACAMBA framework provides a set of objects that implement the presented scheduling technique. It also includes a set of support objects like charts and a thread-safe unification algorithm. CaLi is an instance of a MACAMBA application that implements a Chart parser for the LinGO grammar. The design of CaLi was based on PET (Callmeier, 2000), one of the fastest parsers for LinGO. It implements the quick check and the rule filter, which together take care of filtering over 90% of the failing unification tasks before they are put on the agenda. Deltra is an instance of a MACAMBA application that implements a chart parser for the Deltra grammar. It uses the **DD** parsing schema and uses equality checks to verify the uniqueness of items.

MACAMBA allows the results of parsing runs to be recorded in several ways. Firstly, MACAMBA allows results to be printed to the console. Secondly, it provides an interface to [incr tsdb()] (Oepen, 2001), an application interface to grammar processing components that can collect results from different parsers and provides various tools for analysis of the results. Thirdly, MACAMBA provides an Enterprise Objects[5] relational database interface, which is currently set up to interface with Microsoft Access. To support compatibility with platforms that do not support the Enterprise Object libraries, MACAMBA allows the results to be recorded in an intermediate file that can later be loaded into the database using a special tool. MACAMBA, CaLi, and Deltra were all implemented in Objective-C and currently run on Windows NT, Linux, and Solaris.

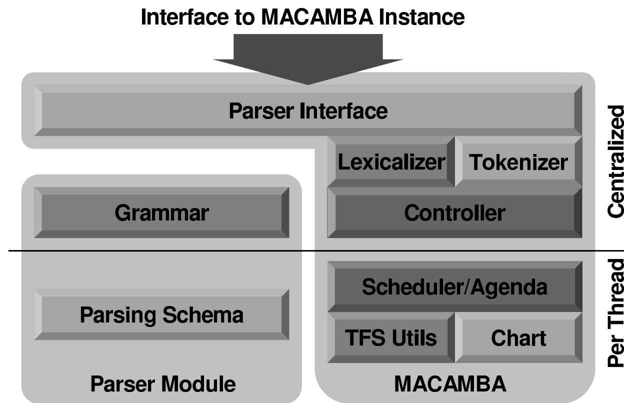


Figure 7.5 Components of a MACAMBA application.

7.3.1 Overall architecture

Figure 7.5 shows a diagram of the most important components in the MACAMBA system. It shows the objects that are provided by the MACAMBA system and objects that should be implemented for each parser instance. MACAMBA specifies protocols to which these objects should conform.[6]

The tokenizer class allows a raw input string to be split into separate tokens. The lexicalizer basically provides a lookup functionality to map tokens to dictionary entries. The abstract lexicalizer class defines a fixed interface for converting tokens to dictionary entries. Concrete subclasses of the lexicalizer can, for example, provide generic constructs for storing entries in memory or provide a generic database interface. Note that the format of dictionary entries typically differs for different grammar formalisms. The abstract lexicalizer class is specified such that the dictionary entries can be of any format.

In order for a parser instance to make use of MACAMBA's multi-threading capabilities, the unification algorithm should be thread safe. By using the unification algorithms provided by MACAMBA, a parser instance automatically complies to this requirement. Nevertheless, it is also possible for a parser instance to provide its own implementation of a unification algorithm.

The parser interface object provides the interface to the rest of the objects embedded in a parser instance. It incorporates the necessary control to setup and manage both the MACAMBA objects and objects specific to the parser instances. It also allows any result or statistic to be accessed through a common interface. A value for a certain statistic can simply be obtained by passing its name to the parser interface. The parser object will use the Objective-C runtime to find the object that implements the respective method for the statistic and return its value. This allows both the user and the designer of a parser in-

[5]The Enterprise Objects libraries are part of the Apple WebObjects development environment.

[6]The protocol construct in Objective-C corresponds to the interface construct in Java.

stance to respectively retrieve and provide statistics without having to know the details of the MACAMBA architecture.

The parser definition file represents the glue for a MACAMBA application. It allows the specification of, for example, which controller, agenda, chart, tokenizer, lexicalizer, grammar, and parsing schema classes should be used. In addition, each object in the system, including user defined objects, can define parameters that can be set in the definition file. MACAMBA will take care that the parameters will be passed to the respective objects.

7.3.2 Parsing Process

Both the agenda and chart classes are part of MACAMBA. We therefore need to explain the interaction between the the agenda and chart objects and the parsing schema object, which is the part of a parser instance. Firstly, a chart in MACAMBA is defined as a repository for items where items are allowed to be grouped per category $c \in \Upsilon$. An item can belong to multiple categories. As discussed in Section 2.3.2, the categories of an item should be chosen such that the set of items traversed in the matching phase is as small as possible.

In MACAMBA, the agenda is a part of the scheduler. Each thread is assigned a scheduling object. When an item is about to be added to the chart, the scheduler will first query the parsing schema class for the categories of the item, using a method $\textsc{Cat} : \mathcal{I} \to \Upsilon^+$. This tells the scheduler to which categories in the chart the item should be added. For each of the categories, it will call a method $\textsc{MatchCat} : \Upsilon \to (\Upsilon \times \mathcal{O})^+$ of the parsing schema class to find out against which sets of items in the chart the item should be matched. The operation type $o \in \mathcal{O}$ specifies which corresponding methods of the parsing schema to use for the match. The scheduler will enumerate over each set,[7] match each item with the newly derived item, and put any resulting task on the agenda. A task will only be added to the agenda if the filter method corresponding to the operation type succeeds.

The methods for the filters, derivation rules, and categorizing of items are all defined in a protocol for parsing schema objects. To allow full flexibility, all details of how the grammar is used in the deduction rules is incorporated in the parsing schema object. An instance of a MACAMBA application is free to define its own subclasses of the agenda, chart, tokenizer, lexicalizer, or any other of the objects.

7.4 Theoretical Analysis

In this section, we will show that the proposed scheduling technique is optimal with respect to the running time, that is, we will show that the running time satisfies equation 3.3. We will also argue that the amount of space that is used by a parallel chart parser roughly equals that of the sequential version.

To prove that the running time of our scheduler is bounded by $O(T_1/P) + T_\infty$, we will show our scheduler is an instance of a greedy scheduler. This basically, means we will show that all threads will always process work as long as work is available at any of the

[7] For Deltra the item and set pair are put on the agenda, and the expansion is postponed.

threads. To this extent, we need to consider the delay that can be caused by the randomized stealing algorithm. There are two factors that determine the success of a steal attempt. First, a thread should select a non-idle thread to steal from. Because a thief is not allowed to select itself, the probability of selecting a victim with work is 1 in case there is only one thief. Therefore, as long as sufficient work is available, it is unlikely that there will be many thieves. Second, the locking attempts should be successful. Because thieves do not wait for locks of victims to be released, it is possible for the victim to thwart a stealing attempt of the thief. Nevertheless, considering that a thief attempts to steal work from more than one queue, the probability of a thief being continually thwarted is very small. In the further discussion, we will assume that work can be stolen in $O(1)$ time.

Theorem 7.3 *On a computer with P processors, algorithm* SCHED *will run any parse represented by a task dependency graph $\langle \mathcal{T}, E, W \rangle$ in $O(T_1/P) + T_\infty$ time, assuming that* STEAL *will steal work in $O(1)$ time.*

Proof. We show that SCHED falls into the class of greedy schedulers, and that the scheduling algorithm itself does not incur more than a constant overhead. To prove SCHED is a greedy scheduler, we need to show that no thread can be idle as long as there is any thread that still has queued work. If a thread t becomes idle while there are still other threads with work, STEAL is given to steal at least some of this work in constant time. In this case, t will resume processing the innermost loop within one iteration of the outermost loop. Since STEAL itself steals in constant time, the total time c_S for stealing work after becoming idle is constant.

Now, STEAL will ensure that the thief will execute at least one of the stolen tasks. This means that the total number of successful steals is bounded by the number of tasks. The total amount of overhead from stealing is therefore bounded by $c_S|\mathcal{T}|$. In addition, each iteration of the innermost loop incurs an additional overhead of c_I. Each iteration, at least one task is executed. Therefore, the total overhead of this loop is bounded by $c_I|\mathcal{T}|$.

The total running time is now bounded by the running time for greedy schedulers plus the overhead: $\sum_{t \in \mathcal{T}} (c_I + c_S) W(t)/P + T_\infty = (c_I + c_S) \sum_{t \in \mathcal{T}} W(t)/P + T_\infty = O(T_1/P) + T_\infty$. □

For the analysis of the space requirements, we consider both the size of the chart and the size of the agenda. The space complexity is typically expressed in relative terms compared to the one processor case. We denote the total space used in the one processor case as S_1 and the combined size in the P processor case as S_P. In both cases, we denote the size of the chart or agenda alone as respectively S_P^C and S_P^A, where P is the number of processors. Finally, we use $S_{P,i}$ to denote the space required by processor i in the P processor case, where $S_P = \sum_{i=1,\ldots,P} S_{P,i}$.

Lemma 7.4 *The total size of the chart S_P^C for a parsing run using* SCHED*, where P is the number of processors, is equal to S_1^C.*

Proof. With SCHED, a derived item is only stored by one thread. Since all items are recorded, it trivially follows that the total space required to store all items is independent of the number of processors. □

The minimum space required for the agenda is determined by the maximum number of tasks that are on the agenda at any point in time during parsing. This number greatly depends on the order of evaluation of the agenda entries. Since the order of evaluation typically varies for differing number of processors—or even between different P processor runs—it is hard to make precise claims about the total space required in such scenarios. We can, however, make such claims if we put restriction on the order of evaluation. Consider an agenda with a depth-first, or stack-based, evaluation strategy.

Lemma 7.5 *When* SCHED *is used for a P processor run in combination with a parser using an agenda with a stack-based order of evaluation policy, the total space S_P^A required for the agenda is bounded by $P \cdot S_1^A$, where S_1^A is the space required in the one processor case.*

Proof. Because the search space will be traversed in a depth-first order, the size of the stack in the one processor case will be bound by the derivation tree depth d (which was defined as the number of match tasks on the critical path). Obviously, in a multiprocessor setup, no processor can exceed this bound. In the worst case, however, each processor may reach this bound. □

Theorem 7.6 *When* SCHED *is used for a P processor run in combination with a parser using an agenda with a stack-based order of evaluation policy, the total space S_P is bounded by PS_1, where S_1 is the space required in the one processor case.*

Proof. Follows trivially from Lemma 7.4 and 7.5. □

As the size of the chart typically dominates the space requirements, however, the space requirements will typically be closer to $O(S_1)$.

7.5 Empirical Analysis

As we saw in Chapter 6, the performance of the sequential version of CaLi is comparable to that of PET. In addition, measurements show that the single-processor parallel version of CaLi incurs an overhead of less than 1% relative to the sequential version of CaLi.

The first set of experiments consisted of running the fuse test suite on a SUN Ultra Enterprise with 8 nodes, each with a 400 MHz UltraSparc processor with 8 MB cache, for a varying number of processors. Table 7.1 shows the results of these experiments.[8] The execution times for each parse are measured in wall clock time. The time measurement of a parse is started before the first thread starts working and ends only when all threads have stopped. A disadvantage of using wall clock time is that it also measures the time when a thread was not running, for example, when it was swapped out by the operating system. It is not possible, however, to measure cpu cycles in a similar fashion. In addition, measuring cpu cycles on a per thread basis yields unreliable results, as measuring on a per thread basis does not guarantee that the recorded work was actually executed simultaneously.

[8] Because the system was shared with other users, only 6 processors could be utilized.

P	T_P (s)	speedup
1	1599.8	1
2	817.5	1.96
3	578.2	2.77
4	455.9	3.51
5	390.3	4.10
6	338.0	4.73

Table 7.1 Execution times for the fuse test suite for various number of processors.

The fuse test suite contains sentences of various complexity, including a lot of small sentences that are hard to parallelize. Table 7.1 shows that even under these conditions it is still possible to obtain a considerable overall speedup.

The second set of experiments were run on a SUN Ultra Enterprise10000 with 64 200 MHz UltraSparc processors and 10GB of main memory. To limit the amount of data generated by the experiments, and to increase the accuracy of the measurements, we selected a subset of the sentences in the fuse suite. The parser is able to parse many sentences in the fuse suite in not much more than a couple of milliseconds. Measuring speedup on a per sentence basis is inaccurate in these cases. We therefore eliminated such sentences from the test suite. From the remaining sentences we made a selection of 500 sentences of various lengths.

The results are shown in Figure 7.6. The figure includes a graph for the maximum, minimum, and average speedup obtained over all sentences. The maximum speedup of 31.4 is obtained at 48 processors. The overall peak is reached at 32 processors where the average speedup is 17.3. Analysis of the measurements indicate that the drop in speedup after 32 processors is caused by overhead in the scheduling algorithm. The minimum speedups of around 1 are obtained for sentences (often short ones) that contain too little inherent parallelism to be parallelized effectively. Table 7.2 shows the average execution times grouped by input length.

Figure 7.6 also shows a graph of the parallel efficiency, which is defined as speedup divided by the number of processors. The average efficiency remains close to 80% up until 16 processors. Note that super linear speedup is achieved with up to 12 processors, repeatedly for the same set of sentences. Super linear speedup can occur because increasing the number of processors can also improve cache behavior. As more processors are used, the amount of data handled by each node reduces. This reduces the chance of cache misses.

In conclusion, we showed that there is sufficient parallelism in parsing computations and presented a parallel chart parser for LinGO that can effectively exploit this parallelism and so achieve considerable speedups. Also, the presented techniques do not rely on a particular parsing schema or grammar formalism, and can therefore be useful for other parsing applications. We therefore believe that the presented techniques provide a good solution to improve the performance of applications where responsiveness is crucial.

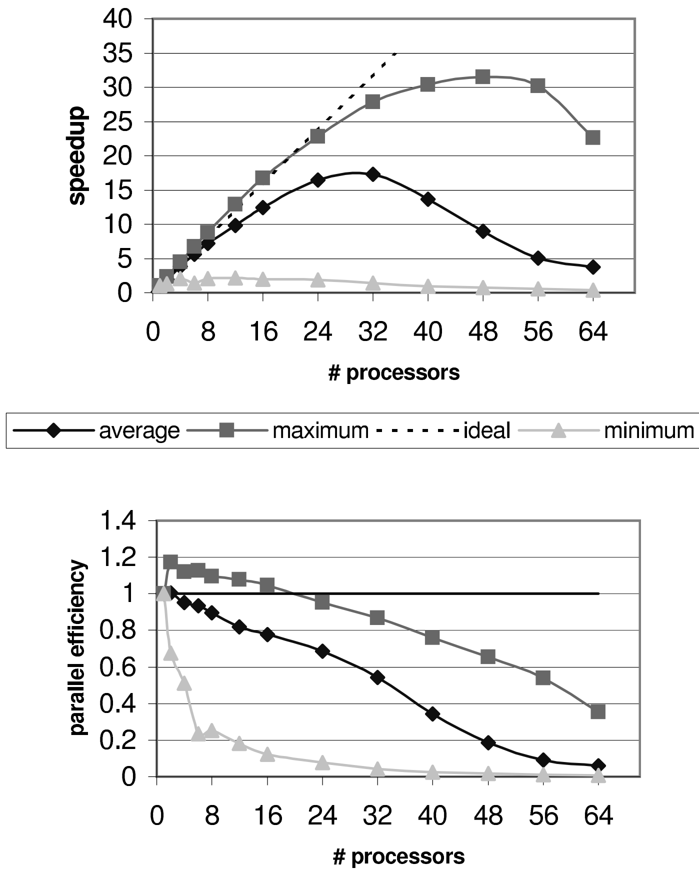


Figure 7.6 Average, maximum, and minimum speedup and parallel efficiency based on a test set of 500 sentences.

	sentence lengths									
	1-10		11-20		21-30		30+		all	
P	$T_P(s)$	S	$T_P(s)$	S	$T_P(s)$	S	$T_P(s)$	S	$T_P(s)$	S
1	34.8	1.0	1465.4	1.0	704.0	1.0	107.3	1.0	2311.5	1.0
2	16.9	2.1	729.2	2.0	348.6	2.0	54.9	2.0	1149.6	2.0
4	9.4	3.7	383.8	3.8	183.3	3.8	29.3	3.7	605.8	3.8
6	6.6	5.3	261.8	5.6	122.4	5.8	19.9	5.4	410.7	5.6
8	5.2	6.7	203.7	7.2	97.7	7.2	17.5	6.1	324.2	7.1
12	5.5	6.4	149.6	9.8	69.9	10.1	13.5	8.0	238.4	9.7
16	3.6	9.7	113.3	12.9	63.7	11.1	12.5	8.6	193.0	12.0
24	3.5	9.9	89.6	16.3	38.5	18.3	6.6	16.2	138.2	16.7
32	5.7	6.1	83.7	17.5	43.8	16.1	5.6	19.0	138.9	16.6
40	10.9	3.2	259.4	5.6	49.9	14.1	5.9	18.3	326.0	7.1
48	10.9	3.2	207.4	7.1	80.6	8.7	9.5	11.3	308.5	7.5
56	11.7	3.0	308.1	4.8	125.2	5.6	9.7	11.1	454.6	5.1
64	19.6	1.8	404.0	3.6	158.8	4.4	12.7	8.4	595.1	3.9

Table 7.2 Execution times for the 64 processor experiments, grouped by input length.

Chapter 8

Optimizing Cache Performance

In this chapter, we explore the possibilities for optimizing cache performance of parsers. Improving the cache performance reduces the stress on the memory bus. This can be crucial in obtaining the desired performance for multiprocessor implementations. In Chapter 6, we already showed how to reduce the memory requirements for unification algorithms, and showed how this lead to improved cache utilization for the Deltra parser. In this chapter, we will show that optimizing cache performance can reduce the number of interference misses further by 50%.

8.1 Introduction

For many sequential applications, improving cache utilization can considerably increase performance. However, efficient use of the cache can even be a more important issue for applications for shared-memory multi processors. The memory bus typically has a limited bandwidth, independent of the number of processors. When too many processors want to communicate with the shared memory simultaneously, the bus can become congested and the processors will be stalled. Improving the cache utilization of processors can mitigate this problem, as processors will be able to perform computations locally in cache more frequently.

We will focus on improving the cache behavior of the Deltra parser. Initial experiments showed that CaLi already achieved low cache miss rates. One of the reasons for this could be the high percentage of structure sharing that is obtained. With the Deltra parser, on the other hand, cache utilization leaves sufficient room for improvement. We will present two approaches to improving cache behavior of Deltra, each of which we will briefly discuss next.

Unification Algorithms The choice of unification algorithm can have a significant impact on the memory bus utilization. In Chapter 6, we introduced two unification algorithms that considerably reduced memory utilization of feature graphs by separating the scratch fields from the graph structures. Obviously, a reduction of the graph size immediately reduces the bus traffic required to read a graph from memory. In addition, separating the scratch fields from the feature structures makes graphs read-only. Temporary changes are recorded in a dedicated buffer. This has the effect of reducing write back traffic.

To improve the spatial reuse of parsers, we also need to look at the feature structure representation. The size of a node of the feature structure used in the unification algorithms presented in Chapter 6 is typically much smaller than the size of a cache line, while the

size of a complete feature structure typically exceeds this size. Spatial reuse could typically be improved by storing the data of a feature graph in the order in which it is typically referenced by the unification algorithm. In this chapter, we will investigate this possibility.

Order of Evaluation Parsing of natural language typically requires vast amounts of memory. In addition, it often occurs that the same set of items is visited repeatedly and in the same order. The combination of these two facts makes parsing applications a candidate for suffering from capacity misses.

Consider the following simplified example. Suppose we are matching each item in a set of newly derived items with the same set of existing items. The newly derived items are stored in a contiguous chunk of memory of size $m\mathcal{C}/a$, where $\mathcal{C}$ is the cache size, a the associativity of the cache, and m a positive number. In addition, the set of existing items are stored in a contiguous chunk of memory of size $n\mathcal{C}/a$, where $n > a$. We assume that each chunk of memory of size $\mathcal{C}/a$ stores β items. Finally, we assume that when iterating over a set, all items in the set will be processed in increasing order of memory address and that each reference of an item requires the entire item to be loaded. Consequently, the number of compulsory misses is equal to the number of cache lines in both chunks of memory: $(m+n)\mathcal{C}/a\mathcal{L}$, where $\mathcal{L}$ is the cache line size.

In the presented scenario, a straightforward control structure can easily lead to capacity misses. Suppose we process all matches for each newly derived item at once. This means we will iterate $m\beta$ times over the set of existing items. Since the set of existing items is accessed in increasing order and given that the referenced memory is at least $(a+1)\mathcal{C}/a$ bytes, the first items of the set will have been completely flushed out of the cache by the time the entire set has been referenced. In other words, the entire set has to be reloaded into the cache on each iteration. Hence, the total number of capacity misses will be $(m\beta - 1)n\mathcal{C}/a\mathcal{L}$.[1]

As we discussed in Section 3.4.2, blocking is an effective technique to reduce capacity misses. In the presented example, a possible blocking strategy would be to iterate over the existing items per block of $b\beta$ items at a time, where $b < a$. Obviously, in this scenario, we eliminate all capacity misses resulting from iterating over the set of existing items. On the other hand, we will need to iterate over the set of newly derived items $\lceil \frac{n}{b} \rceil$ times. Assuming that $m > (a-b)$—guaranteeing that the newly derived items are flushed after processing each block—the number of interference misses is now $m(\lceil \frac{n}{b} \rceil - 1)\mathcal{C}/a\mathcal{L}$. In this particular example, blocking can reduce the total number of misses with approximately a factor of βb. Suppose $m = n = 6$, $\beta = 20$, and $a = 4$. In this case, choosing $b = 3$ leads to a 119 times reduction in the interference misses.

Obviously, the example sketches a highly idealized example and merely serves the purpose of clarifying the problem. Although items belonging to the same set are often stored in increasing order of memory address, they are typically not stored in a contiguous chunk of memory. In addition, feature structures are often not required to be completely loaded into memory. In Section 8.2, we will present a more realistic model.

[1] We do not count the misses that were already counted as compulsory misses.

8.2 Model

In the blocking example we assumed that all items belonging to the same set are stored contiguously. In practice, this is not the case. As we discussed in Section 2.3.2 and 7.3.2, items can belong to multiple sets, or categories. In addition, it is simply often more convenient to store all items in the same memory region. It is, however, safe to assume that all items of a set can be accessed in increasing order of memory. For now, we will also assume that all data associated with an item will be loaded into cache each time it is referenced.

In this section, we will present a model that helps in the understanding of how the non-contiguous storage of items affects a possible blocking strategy. The approach to modeling cache performance was influenced by the research of Harper et. al. (1998, 1999), which presented a convenient way of counting cache misses for loop nests over structured arrays.

8.2.1 Memory Layout

In the example of blocking presented in the previous section, we expressed the size of a set of items in multiples of $\mathcal{C}/a$, where $\mathcal{C}$ was the size and a the associativity of the cache. A block of size $\mathcal{C}/a$ is the largest possible contiguous block of memory such that all memory lines map to a different cache line set in the cache. This characteristic makes it a convenient measure for analyzing cache performance. Note that each block of size $\mathcal{C}/a$ contains $\mathcal{C}/a\mathcal{L}$ memory lines, where $\mathcal{L}$ is the cache line size.

To model the memory reference patterns that result from iterating over a specific set of items, we still assume the set is stored in a number of blocks of size $\mathcal{C}/a$. However, we associate each block with a **footprint** $\psi \in \{0,1\}^{\mathcal{C}/a\mathcal{L}}$, which indicates, for each cache line in the respective block, whether it will be referenced (1) or not (0). We write $\psi(i)$ to reference the ith cache line of a footprint ψ. In addition, we write Ψ_n as a shorthand for a sequence $\psi_1 \ldots \psi_n$ of footprints. A subsequence $\psi_a \ldots \psi_b$ of a sequence of footprints Ψ_n is denoted $\Psi(a,b)$.

Using this model, the memory that is referenced by a set of items can be represented by a sequence of footprints. By counting the number of referenced cache lines, we can easily derive the number of compulsory misses that result from iterating over the set. Given a sequence of footprints Ψ_n, the total number of compulsory misses $\mathcal{S}(\Psi_n)$ is defined as $\sum_{i=1}^{n} \sum_{j=1}^{\mathcal{C}/a\mathcal{L}} \psi_i(j)$.

A sequence of footprints can also be used to determine the number of interference misses after each repeated iteration by counting the number of cache lines that map to sets in the cache with more than a elements. To this extent, we define the function $\Phi : (\{0,1\}^{\mathcal{C}/a\mathcal{L}})^{+} \rightarrow \mathbb{N}$ as follows

$$\Phi(\Psi_n) = \sum_{i=1}^{\mathcal{C}/a\mathcal{L}} \begin{cases} \sum_{j=1}^{n} \psi_j(i) & \sum_{j=1}^{n} \psi_j(i) > a \\ 0 & \text{otherwise.} \end{cases} \tag{8.1}$$

Φ takes as an argument a sequence of footprints and returns the number of interference misses that would result after each repeated iteration over the set represented by the footprints. Obviously, $\Phi(\Psi_n) \leq \mathcal{S}(\Psi_n)$ holds. Note that this model counts both capacity and

conflict misses. In the following discussion we will often not differentiate between these types of misses and simply refer to them as interference misses.

In parsing, a loop over a set of items typically involves another source of data, e.g. another item against which the items in the set are matched. Because this will be loaded into the cache as well, looping over a set of items can yield more interference misses than are given by Equation 8.1. The additional interference misses are given by a function $\Phi : (\{0,1\}^{\mathcal{C}/a\mathcal{L}})^+ \times (\{0,1\}^{\mathcal{C}/a\mathcal{L}})^+ \rightarrow \mathbb{N}$, which is defined as follows.

$$\Phi(\Psi_n, \Psi'_m) = \sum_{i=1}^{\mathcal{C}/a\mathcal{L}} \begin{cases} \sum_{j=1}^{n} \psi_j(i) & \sum_{j=1}^{n} \psi_j(i) \leq a \text{ and} \\ & \sum_{j=1}^{n} \psi_j(i) + \sum_{j=1}^{m} \psi'_j(i) > a \\ 0 & \text{otherwise} \end{cases} \tag{8.2}$$

Since the purpose of the analysis is only to derive the number of interference misses when loading Ψ, we do not count any possible misses resulting from loading the data represented by Ψ'; we require cache misses resulting from loading Ψ' to be counted separately. In addition, we do not count any of the misses that were already counted in Equation 8.1. This strict separation gives us a more precise idea of which effects cause interference misses. More importantly, it allows us to eliminate terms if certain constants, such as the block size in a blocking strategy, are chosen properly.

In the remainder of the discussion we will assume that the associativity a of a cache is greater or equal to 2. In addition, we will assume that all items fall within the boundaries of a footprint and that no item consumes more space than $\mathcal{C}/a$. These assumptions greatly simplify the model.

8.2.2 Cache Miss Equations

Given the representations of sets of items in memory, we can investigate the effect of applying a blocking strategy. Suppose we have two sequences of items, $\mathcal{I}$ and $\mathcal{J}$, each containing respectively $|\mathcal{I}|$ and $|\mathcal{J}|$ items. The items in each of the sequences are sorted in increasing order of memory address. All items of sequence $\mathcal{J}$ need to be matched with the items of sequence $\mathcal{I}$. Both sequences are associated with a sequence of footprints, respectively $\Psi_n^{\mathcal{I}}$ and $\Psi_m^{\mathcal{J}}$, to represent the storage pattern of the items in memory. In addition, we use the notation $\Psi^{\mathcal{I}(i)}$ to indicate the sequence of footprints corresponding to the ith item of sequence $\mathcal{I}$.

Analogous to the example given in the introduction, we will consider a simple loop nest and a loop nest with blocking. Both strategies are shown in Figure 8.1. In the conventional approach, the outermost loop iterates over the items of $\mathcal{J}$ and the innermost loop iterates over the items of $\mathcal{I}$. In the blocking approach, we process the items of sequence $\mathcal{I}$ per b blocks of size $\mathcal{C}/a$. Obviously, in both cases the total number of compulsory misses for performing all matches is simply given by $\mathcal{S}(\Psi_n^{\mathcal{I}}) + \mathcal{S}(\Psi_m^{\mathcal{J}})$.

To count interference misses we should evaluate for each reference of a cache line whether it will cause a hit or a miss. We do not need to count any misses for the items of $\mathcal{J}$, because we can safely assume that each item in set $\mathcal{J}$ will remain in cache until all operations on the

Conventional	**Blocking**

for $j = 1 \ldots \lvert\mathcal{J}\rvert$ **do**
for $i = 1 \ldots \lvert\mathcal{I}\rvert$ **do**
$P_{\mathcal{I}(i),\mathcal{J}(j)}$.
	for $k = 1 \ldots \lceil \frac{n}{b} \rceil$ **do**
	for $j = 1 \ldots \lvert\mathcal{J}\rvert$ **do**
	for $i = b\beta(k-1)+1 \ldots b\beta k$ **do**
	$P_{\mathcal{I}(i),\mathcal{J}(j)}$.

Figure 8.1 Conventional loop nest and equivalent loop nest with blocking, assuming that each block contains β items.

item have completed.[2] The misses resulting from referencing the items in $\mathcal{I}$ is given by

$$(|\mathcal{J}|-1)\Phi(\Psi_n^{\mathcal{I}}) + \sum_{i=1}^{|\mathcal{J}|-1} \Phi(\Psi_n^{\mathcal{I}}, \Psi^{\mathcal{J}(i)}). \tag{8.3}$$

The first term corresponds to the result of the simplified blocking example given in the introduction. Because we now allow for non-contiguous storage of items it is possible that items from $\mathcal{I}$ remain in cache even when $n > a$. We therefore need a second term to count the cases where interference misses are caused by loading items other than those of $\mathcal{I}$. For large n we may suspect that $\Phi(\Psi_n^{\mathcal{I}}) \approx \mathcal{S}(\Psi_n^{\mathcal{I}})$. In this case, the second term does not contribute to the total number interference misses.

The interference misses for the blocking strategy can be obtained analogously. Assuming that $b < a$ the total number of interference misses caused by the presented blocking strategy is given by

$$(\lceil \frac{n}{b} \rceil - 1)\Phi(\Psi_m^{\mathcal{J}}) + \sum_{i=1}^{\lceil \frac{n}{b} \rceil - 1} \Phi(\Psi_m^{\mathcal{J}}, \Psi^{\mathcal{I}}(b(i-1)+1, bi)). \tag{8.4}$$

The first term corresponds to number of interference misses obtained with the blocking strategy of the example given in the introduction. Also in this case the second term can be neglected when $\Phi(\Psi_m^{\mathcal{J}}) \approx \mathcal{S}(\Psi_m^{\mathcal{J}})$.

If we reverse the two loops of the conventional loop nest, the first term of the cache miss equation becomes $(|\mathcal{I}|-1)\Phi(\Psi_m^{\mathcal{J}})$. Comparing this to the first term obtained with the blocking strategy, we can easily see that blocking can yield a $|\mathcal{I}| - 1 : \frac{n}{b} - 1$ improvement over a conventional strategy, and that the blocking argument is still valid for iterating over non-contiguous chunks of data, provided that m and n are sufficiently large. The effectiveness of blocking is mainly reduced by the fact that non-contiguous storage of items reduces the number of items per block.

[2]Under the assumption that $a \geq 2$.

8.2.3 Choosing the Right Value for *b*

One possible way to reduce the term $(\lceil \frac{n}{b} \rceil - 1)\Phi(\Psi_m^{\mathcal{J}})$ is to increase b beyond a. If there are relatively large gaps between the successively stored items, then choosing $b < a$ can yield a waste of cache capacity. In such a case, choosing b to be slightly larger than a will hardly yield additional interference misses.

Consider the blocking strategy presented in the previous section. If we allow b to be greater or equal to a, we need to introduce the following additional terms.

$$\sum_{i=1}^{\lceil \frac{n}{b} \rceil} \left((|\mathcal{J}| - 1)\Phi(\Psi^{\mathcal{I}}(b(i-1)+1, bi)) + \sum_{j=1}^{|\mathcal{J}|-1} \Phi(\Psi^{\mathcal{I}}(b(i-1)+1, bi), \Psi^{\mathcal{J}(j)}) \right) \tag{8.5}$$

These terms correspond to the interference misses incurred by the innermost loop of the blocking strategy. Obviously, if $b = a$ the first term will be zero. Note that if b is chosen to be n, the sum of Equation 8.4 and 8.5 reduces to Equation 8.3.

To give an idea of how choosing $b \geq a$ influences the number of interference misses, we give the following example. Consider a sequence of footprints Ψ_n, where where the probability that a cache line will be referenced is uniformly distributed with a probability of p. We assume that all events of referencing cache lines are independent of each other. We want to know the expected value for $\Phi(\Psi_n)$, assuming that $n > a$. We define X to be the variable indicating the number of memory lines mapping to some cache line set. Since all probabilities are independent, we are performing a Bernoulli trial. The probability $Pr\{X = k\}$ is therefore given by $\binom{n}{k} p^k(1-p)^{n-k}$. For each cache line set, the expectation is equal to the expectation for the Bernoulli trial, minus the $a+1$ terms for the cases where the number of cache lines that map to the respective set is smaller or equal to a.

$$\sum_{k=a+1}^{n} k \binom{n}{k} p^k(1-p)^{n-k} = np - \sum_{k=1}^{a} k \binom{n}{k} p^k(1-p)^{n-k} \tag{8.6}$$

It holds that,

$$\sum_{k=1}^{a} k \binom{n}{k} p^k(1-p)^{n-k} < a \sum_{k=1}^{a} \binom{n}{k} p^k(1-p)^{n-k} = aPr\{X \leq a\}. \tag{8.7}$$

We can therefore give $np - aPr\{X \leq a\}$ as a lower bound for the expectation value for X. Since we defined all probabilities to be independent, we can obtain the expectation for $\Phi(\Psi_n)$ by multiplying it with the number of cache line sets in the cache. The bounds for $\Phi(\Psi_n)$ are given by the following inequality.

$$\Phi(\Psi_n) > (np - aPr\{X \leq a\})\mathcal{C}/a\mathcal{L}. \tag{8.8}$$

Now, compare the first terms of respectively Equation 8.4 and Equation 8.5. As we mentioned, these are typically the most dominant terms. From Equation 8.5 it is clear that

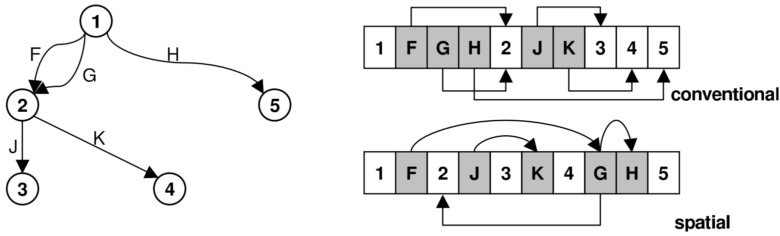


Figure 8.2 Alternative ordering of arcs and nodes of feature structures to improve spatial reuse.

all additional interference misses caused by increasing b are heavily penalized by a factor of $|\mathcal{J}| - 1$. It should be carefully weighed whether these additional misses are compensated by the reduction of the factor $\lceil \frac{n}{b} \rceil - 1$ Obviously, the total benefits can vary considerably from case to case. It should be clear, though, that the possibilities for choosing $b > a$, without increasing the interference misses, are limited.

8.3 Optimizing Spatial Reuse

Because feature structures typically occupy more than one cache line, possibilities for optimizing spatial reuse can best be investigated at the level of the representation of such feature structures.

With Deltra, feature structures are defined to be of a fixed arity, that is, a fixed number of features. A common way to store such feature structures in memory is to store the references of all its arcs right after the node. Most operations on feature structures, however, typically process a graph in a depth first order. The equality check, for example, that is used in Deltra, always processes graphs in a depth first manner. The unification operation largely processes graphs in depth first order unless a subgraph of one graph can be skipped because it can be unified with a variable of the other graph. In a fixed-arity representation, the data of a graph is therefore stored in a different order than typically accessed. In case of a failing unification or equality check, this can lead to redundant loading of arcs.

Suppose for example that a unification fails on the first leaf node of a graph and that four nodes, each with five features, were visited before this leaf node was reached. For each of these four nodes, only one arc was used to reach the leaf node. This means that, in this particular case, sixteen arcs were loaded unnecessarily.

To prevent the redundant loading of arcs, we define an alternative representation where all nodes and arcs are stored in the same order as they are referenced in a depth first traversal. Figure 8.2 shows an example of a graph and both its conventional and alternative representation. With the alternative representation, a depth first traversal will cause nodes and arcs to be referenced in increasing order of memory location. The obvious exception to this rule is when a graph contains reentrancies.

The alternative representation is especially effective in optimizing spatial reuse of an equal-

ity check. As mentioned before, with the unification operation, parts of the graph may be skipped. With equality checks, however, the graphs are guaranteed to be referenced in a depth first manner. Since the equality check is a frequently occurring operation in a typical Deltra parse, we can expect that the alternative ordering can improve spatial reuse.

According to the model presented in the previous section, increasing the spatial reuse can have the additional benefit of increasing temporal reuse. Assuming that a feature structure occupies the same amount of memory regardless of the representation, increasing the spatial reuse typically also reduces the number of cache lines that need be loaded if only part of the graph is referenced. Increasing spatial reuse can therefore lower p and reduce the probability of interference misses occurring.

8.4 Blocking Strategies

In this section, we will present two different blocking strategies. Both strategies exploit the freedom in the order of evaluation of parsing steps to improve cache utilization. The design of both strategies has been kept simple lest the overhead of the implementation defeats the benefits of the improved cache utilization.

8.4.1 Simple Blocking

Deltra uses a single contiguous buffer to store all items. The chart is organized such that items belonging to the same category are linked by a linked list. Because new items are always added to the end of the buffer, items in the linked list are sorted in increasing order of memory address.

A straightforward approach to blocking is to divide the item buffer into fixed size blocks and group operations defined on these blocks. We will call this strategy **simple blocking**. With simple blocking, each block is associated with its own agenda. Obviously, the resulting number of agendas can become large. To keep track of which agendas contain work, a list of non-empty agendas, or an agenda of agendas, needs to be maintained. The order in which non-empty agendas are processed can be of importance. To promote reuse one can, for example, process the agenda with the most queued operations. Note that it is straightforward to define a block size such that the block in which an item is stored can be derived from the item's address.

Obviously, a unification operation is defined on two input graphs. When the input graphs are stored in two different blocks, we need to make a choice which block to associate the work with. As we discussed in Section 2.3.1, the Deltra parser defines an entry on the agenda as a set of operations where one newly derived item is to be matched and unified with a list of existing items. To avoid having to expand the operations, such an agenda entry can best be associated with the block that contains the first item of the list of existing items. The first items in the list of existing items are likely to be in the same block. When iterating over the list leads to an item outside the block, processing of the rest of the list can be postponed by putting an agenda entry for the remaining work on the agenda of the block in which the respective item is stored.

cache	$\mathcal{C}$	a
a	8K	4
b	16K	2
c	32K	4
d	256K	4

Table 8.1 The four different cache configurations used in our experiments.

8.4.2 Categorized Blocking

A disadvantage of the previous approach is that by grouping operations per block, it is not possible to exploit the empty space between items in one set by using block sizes larger than the cache. Since with simple blocking the operations are not strictly ordered per category of items, it is likely that a large proportion of the items in a block is loaded into cache.

With **categorized blocking** we hope to increase reuse by grouping operations defined on the same set of items. In essence, categorized blocking is very similar to simple blocking. Instead, though, of grouping work per region of the buffer of items, we associate work with a predefined category, such as tabular chart cell or a grammar rule.[3] Similarly to simple blocking, the agenda entries are kept sorted per block.

By grouping operations defined on the same category of items, we automatically exploit the small footprints of each list of items. We still apply blocking in a similar fashion to simple blocking. This prevents capacity misses in case the items in a list follow each other closely in memory. However, whenever new work is assigned to a previously processed block of the same category, we process this work first before processing any other block. Because the blocks are processed per category and can be expected to leave a small footprint on the cache, it is likely that data loaded in previous blocks is still in the cache. Categorized blocking can be said to combine locality grouping with blocking.

Categorized blocking can also have the effect of grouping related items in memory when we use, for example, per rule categories. With Deltra, a considerable amount of the deduction steps involve solely items defined on the same rule. This means that grouping items defined on the same rule also has the effect that items of the same group are stored in memory successively. In turn, this can improve the benefits of blocking.

8.5 Results

The cache performance of different algorithms can be effectively compared by counting the number of interference misses. The interference misses directly correspond to traffic between memory and cache. In addition, by not counting compulsory misses, we ignore the traffic that is irrelevant to cache optimizations.

[3]Categories can be defined analogously with the rule-based and tabular chart cell-based grouping heuristic presented in Section 5.3.

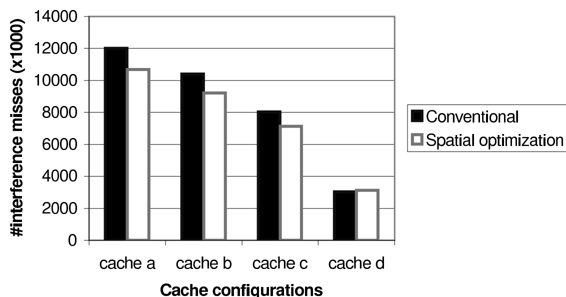


Figure 8.3 Number of interference misses for different feature structure representations and cache configurations.

To measure the interference misses of an algorithm we implemented a configurable cache simulator. The cache simulator simulates a set associative cache with a LRU replacement policy. It can be set to an arbitrary cache size and associativity. Table 8.1 shows the different cache configurations that we have simulated in our experiments. To make the simulation work, the simulator should be told the size and address of the data referenced during processing.

We are only interested in traffic resulting from loading and storing of items and associated feature structures. We therefore only simulate references to the nodes and arcs of the feature structures and other structures used to represent items. Because we ignore the use of other data, such as control structures, the simulation cannot derive all interference misses. However, by doing so, we ensure that the interference misses are a result of the chosen strategy and not an artifact of a specific implementation.

Figure 8.3 shows a comparison between the cache performance resulting from using a conventional feature structure representation and a feature structure representation optimized for spatial reuse. The optimized representation produces slightly larger feature structures. Nevertheless, as can be seen from the figure, the optimized representation can reduce the number of interference misses by over 10%. The best results are obtained with smaller cache sizes. With the larger cache size, the benefits of the optimized representation are diminished. In this case, the slightly increased feature structure size causes it to perform slightly worse. When comparing the execution times for the two algorithms, using an identical setup as with the measurements presented in Section 6.4.2, the spatial optimization resulted in a speedup of several percent. The reduced interference misses could be an explanation for this speedup.

Figure 8.4 shows the results of measuring the interference misses of different evaluation strategies. The results include measurements of an implementation of the simple blocking, categorized blocking, and two conventional approaches, using a FIFO and stack-based agenda. The FIFO and stack-based approach represent, respectively, a breadth-first and depth-first order of evaluation. We also included a version of categorized blocking, where we omitted the blocking by setting the block size to infinite. This allows us to inspect the effect of the locality grouping in isolation.

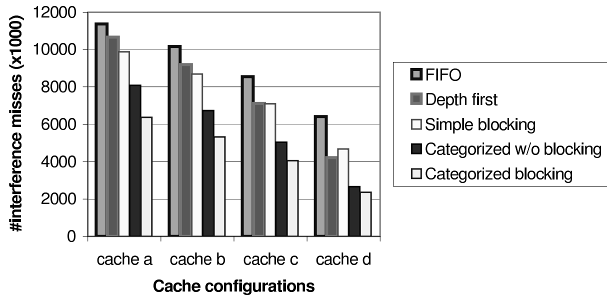


Figure 8.4 Number of interference misses for different evaluation strategies and cache configurations.

The stack-based agenda consistently yields fewer interference misses than the FIFO agenda. A task is typically created as the result of the derivation of a new item. Therefore, at the time of creation of an agenda entry, at least one of the items referenced by the entry typically indicates a recently used item. With the stack-based approach the entries on the agenda are sorted in the order of most recently generated task to least recently generated task. With a FIFO strategy, this order is exactly the reverse. With the stack-based approach, it is therefore more likely that the items involved in a task are still in cache at the time of execution.

With the stack-based approach, capacity misses are still possible. With the simple blocking approach, these effects can be eliminated. On the other hand, the simple blocking strategy may cause a task to be considerably postponed when the task is queued in another block than the block that is being processed at the time of creation. The simple blocking strategy may therefore also cause some reuse to be missed compared to the stack-based strategy. In Figure 8.4, we can see that the simple blocking strategy yields less interference misses than the stack-based strategy for the smaller caches, whereas for larger cache sizes, it performs worse. Obviously, the problem of capacity misses is less urgent for larger cache sizes. This causes the balance to shift for larger cache sizes.

Another problem with the simple blocking strategy is that often no more than two items per block were processed at a time. The memory requirements of Deltra are fairly moderate. In addition, Deltra uses a very fine-grained indexing structure. Consequently, the number of items in a set of items, and the amount of data required to store such a set, is often limited. Obviously, blocking can not be very effective when the number of iterations and the amount of data involved in a loop is limited, because no capacity misses can occur. We expect that simple blocking can still be effective for parsers with a bigger memory footprint.

The categorized blocking strategy proved to be the most effective. Considering differences between the blocked and non-blocked versions, the locality grouping effect has the most influence on the reduction of interference misses. Grouping per category guarantees a certain amount of reuse that is not guaranteed by the stack-based evaluation strategy; sorting tasks in the order of their creation does not guarantee that they will be sorted per item category. Categorized blocking yields over a 50% reduction in interference misses relative to the FIFO strategy. From Figure 8.4 it can be seen that categorized blocking outperformed an identical approach without blocking. The grouping of related operations also promoted a

grouping of related items in memory. This increased the number of items of a set contained per block, which in turn had a positive effect on the effectiveness of the blocking strategy.

Chapter 9

Conclusions and Suggestions for Future Research

In this thesis, we have shown it is possible to considerably speed up the parsing of natural language, with the use of shared-memory multiprocessing capabilities. By a thorough analysis of the parsing process, we were able to identify several limitations for parallel processing. Firstly, we showed that parallelism needs to be exploited at the level of individual unification operations, if unification is taken to be an atomic operation. Secondly, we showed that the total amount of communication can be reduced considerably by applying the right grouping heuristics. We used these results as guidelines for the the design of the parallel parser.

The resulting parallel parser based on LinGO was able to achieve speedups of a factor of over 30 with 48 processors. On average, the maximum speedup was 17 reached at 32 processors. The overall efficiency remains over 80% for up to 16 processors. Overall, compared to the other approaches mentioned in Section 2.4, it seems that the strength of our approach lies in the high efficiency for smaller numbers of processors, high maximum speedups, and ease and flexibility of implementation.

A drawback of our parser is that after the maximum speedup is reached, the speedup drops for an increasing number of processors. In the ideal case, the speedup should remain constant after a maximum speedup is reached. The reason for the drop is an increase in the overhead of the scheduling algorithm for an increased number of processors. One of the problems is that both the number of iterations of the outer loop of our scheduling algorithm (Figure 7.2) and the total number of steal attempts increase for larger numbers of processors. This is especially the case when the total amount of work is limited relative to the number of processors. Obviously, this increases the total work for larger number of processors causing the speedup to decline. Another factor is the fact that each derived item is traversed by each processor, causing an order $O(P)$ overhead per item. Although this overhead is very small, it does contribute to the decrease in speedup for increasing numbers of processors after the maximum speedup is reached.

An interesting topic for future research is to find ways to reduce this decline in speedup for large numbers of processors. One possible approach for limiting the overhead is to reduce the $O(P)$ overhead for checking each item. One possibility is to let each thread handle a particular subset of items such that always only a subset of foreign items need to be checked. An example of such a distribution is to let each thread handle one or more cells of a tabular chart. In this case, a thread would only have to check items in neighboring cells. From the experiments presented in Chapter 5, though, we know that using such a distribution will likely result in more communication. The parallel parser presented by

Ninomiya et al. (2001) provides an example of how a tabular chart can be used to reduce communication. In their approach, the overhead of processing an item is independent of the number of processors. However, it appears as if the overall overhead is larger than with our approach, especially for smaller numbers of processors. For smaller numbers of processors, their parser does not achieve the same levels of efficiency.

Another possible approach is to use a stack-based work stealing approach, such as provided by the Cilk framework (Frigo et al., 1998), in combination with a depth-first search strategy. With the stack-based work stealing approach taken by Cilk, the overhead seems to be less dependent on the number of processors. However, Cilk only guarantees minimal overhead for strict multithreaded computations, as described in Section 3.2.2. Because synchronization of multiple threads through a centralized chart cannot be modeled within a strict multithreaded computation, chart parsers fall outside this class. Consequently, a rather expensive synchronization mechanism is required for the implementation of a shared data structure such as a chart. Another drawback of the stack-based approach is that one does not have the same flexibility in the order of evaluation of tasks as with an agenda-based approach.

Considering the disadvantages of stack-based work stealing approaches for applications like chart parsing, an interesting research topic would be to investigate the usefulness of the scheduling approach presented in Chapter 7 for other applications that are like chart parsing and fall outside of class of strict multithreaded computations. One can think of, for example, applications using a blackboard or memoization techniques. As long as the majority of the tasks can be made thread safe, we suspect that our technique can be useful for exploiting parallelism in such applications.

Another possibility for further improvements of the parser is to further reduce the communication, or bus traffic, required for parsing. One topic of interest is to incorporate the cache optimization, introduced in Chapter 8, into the parallel parser. Another possible research topic is to further optimize the unification algorithm. Suggestions for further improvements on the unification algorithm can be found in Section 6.6.

The results that we have obtained for our parallel parser were solely based on the use of the LinGO and Deltra grammars. Typically, the possibilities for parallelism strongly depend on the specific grammar being used. Although we have limited our research to the LinGO and Deltra grammar, one may expect that the approach may be useful for exhaustive parsing with other grammars. For instance, it is reasonable to suspect that the same approach may work for other HPSG grammars,[1] especially if the size of grammar and dictionary are of the same order of magnitude as those of LinGO. One might also expect, however, that our approach will be useful to parallelize parsers that use other kinds of grammars. Another popular grammar formalism, for example, is LFG. One difference is that typically far fewer unifications fail when a LFG grammar is used. However, we saw that early detection and preventing the execution of failing unifications with the quick check did not thwart the ability to obtain considerable speedups. Really, the only prerequisite for obtaining speedup is that the average parallelism in parsings resulting from using any kind of grammar is sufficiently large. If the average available parallelism is too small, it will not be possible to speed up parsing by means of parallel processing, without resorting to parallel unification. If, on

[1] Remember that LinGO is a HPSG grammar.

the other hand, there is a sufficient amount of parallelism, our scheduling algorithm provides a way of exploiting this parallelism, because the work-stealing algorithm ultimately allows each single unification task to be distributed independently. We hope that the step by step approach from analysis to implementation, as we have presented in this thesis, will prove to be useful as a guideline in accomplishing similar results for other parsers.

Besides researching the applicability of our parallel parsing technique in combination with other grammars, it is also useful to investigate its usefulness for variations of chart parsing. Many of the extensions to sequential chart parsing can be expected to work with our approach without much modification, as long as the added functionality can be processed in a thread-safe manner. For example, the introduction of probabilistic parsing should typically not pose a problem, because the computation of the probabilities is typically a thread safe operation.

One possible complication arises if our technique is to be used for n-first parsing. With n-first parsing, the tasks on the agenda are processed in the order of some assigned priority, e.g. derived from the predicted probability. All our research focussed on exhaustive parsing, where one aims at finding all possible parse trees. With exhaustive parsing, the use of a distributed agenda is a convenient and efficient way of reducing synchronization and communication. With n-first parsing, however, the use of a distributed agenda might pose a problem, because it may cause the tasks to be processed in the incorrect order. This may ultimately lead to the processing of redundant tasks or even the derivation of less likely or less preferable parse trees. In other words, n-first parsing might require the use of a centralized agenda. Parallel n-first parsing remains an interesting topic for future research. Possible research could focus on finding a way to use distributed agendas exploiting the remaining freedom in the order of evaluation. The major goal of such an approach is how to prevent executing tasks that would otherwise never be executed in a sequential setup. Another approach could be the implementation of an efficient centralized chart. Techniques to reduce synchronization overhead in the case of a centralized agenda are discussed in Section 3.2.2.

References

[Adriaens and Hahn1994] Geert Adriaens and Udo Hahn, editors. 1994. *Parallel Natural Language Processing*. Ablex Publishing Corporation, Norwood, New Jersey.

[Aho et al.1974] Alfred V. Aho, John E. Hopcroft, and J. D. Ullman. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading.

[Alshawi1992] Hiyan Alshawi, editor. 1992. *The Core Language Engine*. ACL-MIT press, Cambridge, Mass.

[Amtrup1997] Jan W. Amtrup. 1997. ICE: A communication environment for natural language processing. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, Las Vegas, Nevada, USA., June. IEEE. University of Hamburg, Germany.

[Barnard and Simon1994] Stephen T. Barnard and Horst D. Simon. 1994. Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and Experience*, 6(2):101–117.

[Blumofe and Leiserson1993] Robert D. Blumofe and Charles E. Leiserson. 1993. Space-efficient scheduling of multithreaded computations. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing (STOC '93)*, pages 362–371, San Diego, CA, USA, May. Also in SIAM Journal on Computing.

[Blumofe and Leiserson1994] Robert D. Blumofe and Charles E. Leiserson. 1994. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS'94)*, pages 356–368, November.

[Brent1974] Richard P. Brent. 1974. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, April.

[Bresnan and Kaplan1982] Joan Bresnan and Ronald Kaplan. 1982. Lexical-functional grammar: A formal system for grammatical representation. In Joan Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages 173–281. The MIT Press, Cambridge, MA.

[Briscoe et al.1987] Ted Briscoe, Claire Grover, Bran Boguraev, and John Carrol. 1987. A formalism and environment for the development of a large grammar of english. In John McDermott, editor, *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pages 703–708, Milan, Italy, August. Morgan Kaufmann.

[Callmeier2000] Ulrich Callmeier. 2000. PET – A platform for experimentation with efficient HPSG. *Natural Language Engineering*, 6(1):1–18.

[Carpenter1992] Bob Carpenter. 1992. *The Logic of Typed Feature Structures*. Cambridge University Press, Cambridge, England.

[Carroll1994] John Carroll. 1994. Relating complexity to practical performance in parsing with wide-coverage unification grammars. In *Proc. of the 32nd Annual Meeting of the Association for Computational Linguistics*, pages 287–294, Las Cruces, NM, June27–30.

[Copestake1999] Ann Copestake, 1999. *The (new) LKB system, version 5.2*. CSLI, Stanford University.

[Cowie and Lehnert1996] Jim Cowie and Wendy Lehnert. 1996. Information extraction. *Communications of the ACM*, 39(1):80–91, January.

[Dandamudi1991] S. Dandamudi. 1991. A comparison of task scheduling strategies for multiprocessor systems. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 423–426, December.

[de Vreught and Honig1989] J.P.M. de Vreught and H.J. Honig. 1989. A tabular bottom-up recognizer. Technical Report 90-31, Delft University of Technology, dept. Computer Science, Delft, Netherlands.

[de Vreught and Honig1991] J.P.M. de Vreught and H.J. Honig. 1991. Slow and fast parallel recognition. In *2nd International Workshop on Parsing Technologies (TWLT)*, Cancun, Mexico.

[de Vreught1993] J.P.M. de Vreught. 1993. *Parallel Parsing*. Ph.D. thesis, Delft University of Technology, Delft.

[Dijkstra et al.1983] Edsger W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren. 1983. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16(5):217–219, June.

[Dijkstra1965] E. W. Dijkstra. 1965. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September.

[Ding and Kennedy1999] Chen Ding and Ken Kennedy. 1999. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 229–241.

[Dowding et al.1994] J. Dowding, J. M. Gawron, D. Appelt, and J. Bear. 1994. Gemini: A natural language system for spoken-language understanding. In *Proc. ARPA Human Language Technology Workshop '93*, pages 43–48, Princeton, NJ.

[Dwork et al.1984] Cynthia Dwork, Paris Kanellakis, and John Mitchell. 1984. On the sequential nature of unification. *Journal of Logic Programming*, 1(1):35–50.

[edi1995] 1995. Editorial in natural language engineering 1(1). Cambridge University Press.

[Emele1991] Martin C. Emele. 1991. Unification using lazy nonredundent copying. In *29th. Annual Meeting of the Association for Computational Linguistics*, Berkeley, California. Also available as: Project Polygloss Paper, Institut für Maschinelle Sprachverarbeitung, University of Stuttgart, Germany.

[Flickinger et al.1987] D. Flickinger, J. Nerbonne, I.A. Sag, and T. Wasow. 1987. Toward evaluation of NLP systems. Technical report, Hewlett-Packard Laboratories.

[Frigo et al.1998] Matteo Frigo, Charles E. Leiserson, and Keigh H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices*, 33(5):212–223, May.

[Fujioka et al.1990] T. Fujioka, H. Tomabechi, O. Furuse, and H. Iida. 1990. Parallelization technique for quasi-destructive graph unification algorithm. In *Information Processing Society of Japan SIG Notes 90-NL-80*. In japanese.

[Ghosh et al.1997] S. Ghosh, M. Martonosi, and S. Malik. 1997. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 11th International Conference on Supercomputing (ICS-97)*, pages 317–324, New York, July 7–11. ACM Press.

[Godden1990] Kurt Godden. 1990. Lazy unification. In 28th *Annual Meeting of Association for Computational Linguistics*, pages 180–187, Pittsburgh.

[Görz et al.1996] Günther Görz, Marcus Kesseler, Jörg Spilker, and Hans Weber. 1996. Research on architectures for integrated speech/language systems in Verbmobil. In *The 16th International Conference on Computational Linguistics*, volume 1, pages 484–489, Copenhagen, Danmark, August5–9.

[Graham1969] R.L. Graham. 1969. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17(2):416–429.

[Grishman and Chitrao1988] R. Grishman and M. Chitrao. 1988. Evaluation of a parallel chart parser. In *Second Conf. on Applied Natural Language Processing*, pages 71–76. Association for Computational Linguistics, 9–12 February.

[Hahn1994] Udo Hahn. 1994. An actor model of distributed natural language parsing. In Geert Adriaens and Udo Hahn, editors, *Parallel Natural Language Processing*. Ablex Publishing Corporation, Norwood, New Jersey.

[Harper et al.1998] John S. Harper, Darren J. Kerbyson, and Graham R. Nudd. 1998. Analytical modeling of set-associative cache behaviour. Technical Report CS-RR-349, University of Warwick, October 15,.

[Harper et al.1999] J. S. Harper, D. J. Kerbyson, and G. R. Nudd. 1999. Efficient analytical modelling of multi-level set-associative caches. *Lecture Notes in Computer Science*, 1593:473.

[Hendrickson and Leland1995] Bruce Hendrickson and Robert Leland. 1995. A multi-level algorithm for partitioning graphs. In Sidney Karin, editor, *Proceedings of the 1995 ACM/IEEE Supercomputing Conference, December 3–8, 1995, San Diego Convention Center, San Diego, CA, USA*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA. ACM Press and IEEE Computer Society Press.

[Hoogerbrugge and de Vreught1991] J. Hoogerbrugge and J.P.M. de Vreught. 1991. Parallel recognizing in practice. Technical Report 91-26, Delft University of Technology, dept. Computer Science, Delft, Netherlands.

[Ibarra et al.1991] Ibarra, Pong, and Sohn. 1991. Parallel recognition and parsing on the hypercube. *IEEE Transactions on Computers*, 40(6):764–770.

[JáJá1992] Joseph JáJá. 1992. *An introduction to Parallel Algorithms*. Addison-Wesley.

[Kacsuk1990] Péter Kacsuk. 1990. *Execution Models of Prolog for Parallel Computers*. Research monographs in parallel and dustributed computing. MIT Press, Cambridge, Ma.

[Kaplan1973] R. Kaplan. 1973. A general syntactic processor. In E. Rustin, editor, *Natural Language Processing*. Prentice-Hall, Englewood Cliffs, NJ.

[Karp and Zhang1993] Richard M. Karp and Yanjun Zhang. 1993. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM*, 40(3):765–789, July.

[Karttunen1986] L. Karttunen. 1986. D-PATR: A development environment for unification-based grammars. Technical Report CSLI-86-61, SRI International and Center for the Study of Language and Information.

[Karypis and Kumar1995] George Karypis and Vipin Kumar, 1995. *METIS, Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*. Minneapolis, MN 55455, August.

[Karypis and Kumar1998] George Karypis and Vipin Kumar. 1998. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 10 January.

[Karypis and Kumar1999] George Karypis and Vipin Kumar. 1999. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, January.

[Kasami1965] T. Kasami. 1965. An efficient recognition and syntax algorithm for context-free languages. Scientific Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford MA.

[Kay1980] Martin Kay. 1980. Algorithm schemata and data structures in syntactic processing. In Karen Sparck-Jones Barbara J. Grosz and Bonnie Lynn Webber, editors, *Readings in Natural Language Processing*, pages 35–70. Morgan Kaufmann, Los Altos.

[Kay1985] Martin Kay. 1985. Parsing in functional unification grammar. In Lauri Karttunen David R. Dowty and Arnold M. Zwicky, editors, *Natural Language Parsing*, pages 251–278. Cambridge University Press, Cambridge.

[Kernighan and Lin1970] B. W. Kernighan and S. Lin. 1970. An efficient heuristic for partitioning graphs. *Bell Systems Technical J.*, 49:291–307.

[Kiefer et al.1999] Bernd Kiefer, Hans-Ulrich Krieger, John Carroll, and Rob Malouf. 1999. A bag of useful techniques for efficient and robust parsing. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics (ACL-99)*, pages 473–480, University of Maryland, USA.

[Kogure1990] Kiyoshi Kogure. 1990. Strategic lazy incremental copy graph unification. In *Proceedings of the 13th International Conference on Computational Linguistics COLING-90*, volume 2, pages 223–228, Helsinki, Finland.

[Kruskal and Weiss1985] Clyde P. Kruskal and Alan Weiss. 1985. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, 11(10):1001–1016, October.

[Lozinskii and Nirenburg1986] Eliezer L. Lozinskii and Sergei Nirenburg. 1986. Parsing in parallel. *Computer Languages*, 11(1):39–51.

[Malouf et al.2000] Robert Malouf, John Carroll, and Ann Copestake. 2000. Efficient feature structure operations without compilation. *Natural Language Engineering*, 6(1):1–18.

[Manousopoulou et al.1997] A.G. Manousopoulou, G. Manis, P. Tsanakas, and G. Papakonstantinou. 1997. Automatic generation of portable parallel natural language parsers. In *Proceedings of the 9th Conference on Tools with Artificial Intelligence (ICTAI '97)*, pages 174–177. IEEE Computer Society Press.

[Markatos and LeBlanc1992a] E. P. Markatos and T. J. LeBlanc. 1992a. Using processor affinity in loop scheduling on shared-memory multiprocessors. In IEEE Computer Society. Technical Committee on Computer Architecture, editor, *Proceedings, Supercomputing '92: Minneapolis, Minnesota, November 16-20, 1992*, pages 104–113, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA. IEEE Computer Society Press.

[Markatos and LeBlanc1992b] Evangelos P. Markatos and Thomas J. LeBlanc. 1992b. Load balancing vs. locality management in shared-memory multiprocessor. In *Proceedings of the 1992 International Conference on Parallel Processing*, volume I, Architecture, pages I–258–I–267, Boca Raton, FL, August. CRC Press. Shared-Memory Multiprocessor Trends and the Implications for Parallel Program Performance TR 420 May 1992 $2.00; 28 pages; in pub/papers/systems.

[Matsumoto et al.1983] Y. Matsumoto, H. Tanaka, H. Hirakawa, H. Miyoshi, and H. Yasukawa. 1983. Bup: A bottom-up parser embedded in prolog. In *New Generation Computing*, volume 1, pages 145–158.

[Nederhof1993] Mark-Jan Nederhof. 1993. Generalized left-corner parsing. In *6th Meeting of the European Association of Computational Linguistics (ACL)*, pages 305–314, Utrecht, Netherlands.

[Neuhaus and Hahn1996] Peter Neuhaus and Udo Hahn. 1996. Restricted parallelism in object-oriented lexical parsing. In *Proc. of the 16th Int. Conf. on Computational Linguistics*, Copenhagen, DK, 5–9 Aug.

[Nijholt1991] A. Nijholt. 1991. Overview of parallel parsing strategies. In M. Tomita, editor, *Current Issues in Parsing Technology*, chapter 14. Kluwer Academic Publishers, Norwell, MA.

[Nijholt1994] Anton Nijholt. 1994. Parallel approaches to context-free language parsing. In Geert Adriaens and Udo Hahn, editors, *Parallel Natural Language Processing*. Ablex Publishing Corporation, Norwood, New Jersey.

[Ninomiya et al.1998] Takashi Ninomiya, Kentaro Torisawa, and Jun'ichi Tsujii. 1998. An efficient parallel substrate for typed feature structures on shared memory parallel machines. In *Proceedings of the 17th International Conference on Computational Linguistics and the 36th Annual Meeting of the Association for Computational Linguistics*, pages 968–974, Montreal, Canada, August.

[Ninomiya et al.2001] Takashi Ninomiya, Kentaro Torisawa, and Jun'ichi Tsujii. 2001. An agent-based parallel HPSG parser for shared-memory parallel machines. *Journal of Natural Language Processing*, 8(1), January.

[Nirenburg1995] Sergie Nirenburg. 1995. The Pangloss Mark III machine translation system. Technical Report CMU-CMT-95-145, CMU, CMT.

[Nurkkala and Kumar1994a] T. Nurkkala and V. Kumar. 1994a. The performance of a highly unstructured parallel algorithm on the KSR1. In IEEE, editor, *Proceedings of the Scalable High-Performance Computing Conference, May 23–25, 1994, Knoxville, Tennessee*, pages 215–220, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA. IEEE Computer Society Press.

[Nurkkala and Kumar1994b] Tom Nurkkala and Vipin Kumar. 1994b. A parallel parsing algorithm for natural language using tree adjoining grammar. In Howard Jay Siegel, editor, *Proceedings of the 8th International Symposium on Parallel Processing*, pages 820–829, Los Alamitos, CA, USA, April. IEEE Computer Society Press.

[Oepen and Callmeier2000] Stephan Oepen and Ulrich Callmeier. 2000. Measure for measure: Parser cross-fertilization. In *Proceedings sixth International Workshop on Parsing Technologies (IWPT'2000)*, pages 183–194, Trento, Italy.

[Oepen and Carroll2000a] Stepan Oepen and John Carroll. 2000a. Ambiguity packing in constraint-based parsing - practical results. In *Proceedings of the 1st Conference of the North American Chapter of the Association for Computational Linguistics*, pages 162–169, Seattle, WA.

[Oepen and Carroll2000b] Stephan Oepen and John Carroll. 2000b. Parser engineering and performance profiling. *Natural Language Engineering*, 6(1):81–97.

[Oepen2001] Stephan Oepen. 2001. [incr tsdb()] — competence and performance laboratory. User manual. Technical report, Computational Linguistics, Saarland University, Saarbrücken, Germany. in preparation.

[Olk and de Vreught1992] J.G.E. Olk and J.P.M. de Vreught. 1992. Evaluating master-slave implementation of double-dot parsing for massive parallel mimd systems. Technical Report 92-78, Delft University of Technology, dept. Computer Science, Delft, Netherlands.

[Pereira1985] Fernando C. N. Pereira. 1985. A structure-sharing representation for unification-based grammar formalisms. In *Proc. of the 23rd Annual Meeting of the Association for Computational Linguistics. Chicago, IL, 8–12 Jul 1985*, pages 137–144.

[Pollard and Sag1994] Carl Pollard and Ivan Sag. 1994. *Head-Driven Phrase Structure Grammar*. University of Chicago Press, Chicago. Draft distributed at the Third European Summer School in Language, Logic and Information, Saarbrücken, 1991.

[Polychronopoulos and Kuck1987] Constantine D. Polychronopoulos and David J. Kuck. 1987. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, December. CSRD TR #641 January 1987.

[Pontelli et al.1998] Enrico Pontelli, Gopal Gupta, Janyce Wiebe, and David Farwell. 1998. Natural language multiprocessing: A case study. In *Proceedings of the 15th National Conference on Artifical Intelligence (AAAI '98)*, July.

[Rosenkrantz and Lewis1970] D.J. Rosenkrantz and P.M. Lewis. 1970. Deterministic left corner parsing. In *11th Annual Symposium on Switching and Automata Theory*, pages 139–152.

[Rytter1986] W. Rytter. 1986. On the complexity of parallel parsing of general context-free languages. *Theoretical Computer Science*, 47(3):315–321.

[Sikkel and Nijholt1997] Klaas Sikkel and Anton Nijholt. 1997. Parsing of context-free languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages (Vol 2) Linear Modeling: Background and Application*. Springer Verlag, Berlin.

[Sikkel and op den Akker1996] Klaas Sikkel and Rieks op den Akker. 1996. Predictive head-corner chart parsing. In Harry Bunt and Masaru Tomita, editors, *Recent Advances in Parsing Technology*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

[Sikkel1993a] K. Sikkel. 1993a. On-line parsing in constant time per word. *Theoretical Computer Science*, 120:303–310.

[Sikkel1993b] Klaas Sikkel. 1993b. *Parsing Schemata*. Ph.D. thesis, Dept. of Computer Science, University of Twente, Enschede, The Netherlands.

[Stanfill and Waltz1986] Craig Stanfill and David L Waltz. 1986. Toward memory-based reasoning. *Communications of the ACM*, 29(12):1213–1228.

[Tang and Yew1986] Peiyi Tang and Pen-Chung Yew. 1986. Processor self-scheduling for multiple nested parallel loops. In *Proceedings 1986 International Conference Parallel Processing*, pages 528–535, August.

[Taura and Yonezawa1999] K. Taura and A. Yonezawa. 1999. StackThreads/MP: Integrating futures into calling standards. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[Thompson1991] Henry S. Thompson. 1991. Chart parsing for loosely coupled parallel systems. In M. Tomita, editor, *Current Issues in Parsing Technology*, chapter 15. Kluwer Academic Publishers, Norwell, MA.

[Tomabechi1991] H. Tomabechi. 1991. Quasi-destructive graph unifications. In *Proceedings of the 29th Annual Meeting of the ACL*, Berkeley, CA.

[Tomabechi1992] Hideto Tomabechi. 1992. Quasi-destructive graph unifications with structure-sharing. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING-92)*, Nantes, France.

[Tomabechi1995] Hideto Tomabechi. 1995. Design of efficient unification for natural language. *Journal of Natural Language Processing*, 2(2):23–58.

[Tomita1985] M. Tomita. 1985. *Efficient parsing for natural language*. Kluwer Academic Publishers, Boston, MA.

[van Lohuizen1999] Marcel van Lohuizen. 1999. Parallel processing of natural language parsers. In *Proceedings of the International Conference ParCo99*, pages 168–175, Delft, The Netherlands.

[van Lohuizen2000] Marcel P. van Lohuizen. 2000. Memory-efficient and thread-safe quasi-destructive graph unification. In *Proceedings of the 38th Meeting of the Association for Computational Linguistics*, Hong Kong, China.

[van Lohuizen2001a] Marcel P. van Lohuizen. 2001a. Efficient and thread-safe unification with LinGO. In Stephan Oepen, Daniel Flickinger, Jun-Ichi Tsujii, and Hans Uszkoreit, editors, *Efficiency in Unification-Based Processing*. Center for the Study of Language and Information, Stanford, CA. Forthcomming.

[van Lohuizen2001b] Marcel P. van Lohuizen. 2001b. A generic approach to parallel chart parsing with an application to LinGO. In *Proceedings of the 39th Meeting of the Association for Computational Linguistics*, Toulouse, France.

[Vitter and Simons1986] Jeffrey Scott Vitter and Roger A. Simons. 1986. New classes for parallel complexity: A study of unification and other complete problems for $\mathcal{P}$. *IEEE Transactions on Computers*, C-35(5):403–418, May.

[Wilks1996] Yorick Wilks. 1996. Natural language processing. *Communications of the ACM*, 39(1):60–62, January.

[Wolf and Lam1991] Michael E. Wolf and Monica S. Lam. 1991. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, June 26–28.

[Wroblewski1987] David A. Wroblewski. 1987. Nondestructive graph unification. In Howard Forbus, Kenneth; Shrobe, editor, *Proceedings of the 6th National Conference on Artificial Intelligence (AAAI-87)*, pages 582–589, Seattle, WA, July. Morgan Kaufmann.

[Wu and Kung1991] J.-C. Wu and H. T. Kung. 1991. Communication complexity for parallel divide-and-conquer. In IEEE, editor, *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 151–162, San Juan, Porto Rico, October. IEEE Computer Society Press.

[Yasuura1984] Hiroto Yasuura. 1984. On parallel computational complexity of unification. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'84)*, pages 235–243, Amsterdam. Institute for New Generation Computer Technology [ICOT].

[Yonezawa and Ohsawa1994] Akinori Yonezawa and Ichiro Ohsawa. 1994. Object-oriented parallel parsing for context-free grammers. In Geert Adriaens and Udo Hahn, editors, *Parallel Natural Language Processing*. Ablex Publishing Corporation, Norwood, New Jersey.

[Yoshida et al.1999] Minoru Yoshida, Takashi Ninomiya, Kentaro Torisawa, Takaki Makino, and Jun'ichi Tsujii. 1999. Efficient FB-LTAG parser and its parallelization. In *Proceedings of Pacific Association for Computational Linguistics '99*, pages 90–103, Waterloo, Canada, August.

[Younger1967] Daniel H. Younger. 1967. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, February.

Summary

Parallel Natural Language Parsing: From Analysis to Speedup

Focus of Research

The field of natural language processing aims at giving computers the ability to process or understand human language, be it spoken or written. Applications include machine translation, natural language interfaces, information extraction, and information retrieval. Applications that aim at coming to a full understanding of a given sentence often use unification-based grammar parsing to analyze the input. Such parsers can be seen as context-free parsers augmented with additional constraint checking in the form of unification of features structures.[1] For such applications, parsing is often the most computationally extensive operation. In turn, over 90% of the parsing time is spent performing unifications.

Most natural language applications are designed for single processor systems. The increasing availability of multiprocessor shared-memory systems therefore seems to provide the means to improve the performance of such applications by several factors. In applications where multiple sentences can be processed independently of each other, parallelism can be trivially exploited. We focus on parallelizing the parsing process itself. For application where the processing of a single sentence forms a bottleneck, this is the only way to exploit multi-processor capabilities. Applications that require direct user interaction typically fall into this category.

Chart parsing is one of the most popular techniques for the parsing of natural language. The chart of a chart parser records intermediate results. These results are used both as an efficient way to store multiple parse trees and to prevent duplicating work. Chart parsers often also have an agenda, which holds a list of tasks to be performed by the parser. The parallel parser we will present is based on the chart parsing paradigm.

Initially, our research focussed on developing a parallel parser for the Deltra grammar, developed at the Delft University of Technology. Later we also implemented a parser for the LinGO grammar, from CSLI Stanford. Since the ability to achieve speedup is greatly influenced by the grammar, it is desirable to test the developed techniques in multiple contexts. In addition, the LinGO framework comes with extensive test suites and standardized testing tools that allow for easy performance comparison with other platforms.

Exploring the Feasibility of Parallel Parsing

Implementing a multi-threaded application can be a cumbersome task. It is often possible to derive useful information from its sequential counterpart that can aid in making the

[1] It should be noted though that with modern grammar design there is often no explicit context-free backbone.

right design decisions in an early stage of development. Firstly, we investigated the possibilities for parallelism in parsing. It is known that the running time of any multithreaded application T_P on a P processor system can be bound by the inequality

$$T_P \leq T_1/P + T_\infty,$$

where T_1 is the running time on one processor and T_∞ the critical path, or minimum running time, of an application. Also the maximum speedup that can be obtained for an application can never exceed the average available parallelism T_1/T_P. We derived the values of T_1 and T_∞ for each sentence in the fuse test suite by representing individual parses as a task graph. In such a graph, tasks are represented as nodes, annotated with the time needed to execute the task. The arcs of the graph represent the dependencies. The largest path in such graph equals T_∞.

We investigated two different ways to define the type of work a single tasks executes. For the first type of task graph we defined a task to perform a single unification operation. Since unification is known to be hard to parallelize, it is reasonable to take unification as an atomic operation. For the second type of task graph, we defined a task to perform a number of related unification operations.

The results showed that the average available parallelism was only sufficient in the case where each unification operation was taken as a single task. This implies that a parallel parser can only obtain speedup when it allows each unification operation to be distributed independently.

Another important aspect that determines the feasibility of parallel parsing is the communication that is required between the different processors. Even though all processors of a shared-memory system have direct access to all data, communication still plays a role in the form of cache misses. To limit the complexity of the simulations, however, we used the network model to approximate the communication patterns. Based on the results from the previous experiments, we defined each unification operation to be a different task.

Preliminary experiments showed that arbitrarily assigning tasks to processors can yield excess communication. A solution to this is to group operations using the same data on the same processor. Since determining an optimal distribution of work at run-time is infeasible, we must base such groupings on heuristics. In addition, since we still need to be able to distribute individual unification tasks, such heuristic can only serve as a guideline. A good heuristic will reduce communication while keeping a well-balanced work load.

We tested three different grouping heuristics, which formed groups based on respectively, the range of the input sentence represented by a task, the rule represented by the result of a task, and a greedy approach where operations are performed on the processor that happens to store the required data.

Again, the measurements were based on individual parsings of sentences from the fuse test suite. For each heuristic we computed an arbitrary distribution of work (after grouping the tasks) and an approximation of the best case. The greedy heuristic yielded the best results for the LinGO grammar. It resulted in the most balanced distributions, was effective in reducing the worst-case communication, and allowed for a minimal communication close to the best case without grouping. For Deltra the rule-based approach proved to be useful.

Deltra has more grammar rules than LinGO and uses a different parsing technique for which grouping per rule is very suitable.

Efficient and Effective Parallel Parsing

The first step in developing a parallel parser is finding a thread-safe unification algorithm. Without such an algorithm, it is not possible for one feature structures to be involved in multiple unification operations simultaneously. Such a restriction can make the distribution of unification tasks amongst processors hard, especially when structure sharing is used (which is usually the case).

We present an efficient thread-safe unification algorithm that is based on Tomabechi's algorithm. Tomabechi's algorithm is frequently used, because it is known to be one of the fastest unification algorithms for natural language parsing. A key to the performance obtained with Tomabechi's algorithm is the use of control fields, or scratch fields, incorporated in the nodes of the feature structures. Besides causing the algorithm to be not thread-safe, these scratch fields also have the drawback of unnecessarily occupying memory space when the associated nodes are not involved in a unification operation. Also other popular unification algorithms use scratch fields.

Our main approach to making unification thread-safe is to move scratch fields into separate buffers and associate them with nodes at run time. This way, different threads can associate different scratch fields to the same node. The same feature structure can therefore occur in unification operations performed by different threads simultaneously without any harm. Since scratch fields are only associated with nodes for the duration of a unification operation, they can be reused in successive operations. This approach therefore also eliminates the superfluous memory consumption of the original approach.

We investigated two different strategies for associating scratch buffers with nodes. The most straightforward approach is to use hashing, where the node's physical address is used to associate it with a scratch buffer. The more complicated approach is to associate a unique index into an array of scratch buffers with each node. For each feature structure, the nodes are numbered in a depth-first manner, starting from the root. Because different feature structures typically have common index numbers, each feature structure involved in a unification is associated with an additional offset into the array.

Even though the indexing technique requires additional computations to associate scratch buffers, it proved to be about equally efficient as Tomabechi's algorithm. The reason for this is that our approach results in better cache utilization due to the reusing of the scratch buffers. The indexing technique also proved to be about 10% more efficient than the hashing algorithm. Overall memory consumption for both the hashing and indexing techniques was reduced by roughly 50% for LinGO and roughly 75% for Deltra.

Having a thread-safe unification algorithm, we can proceed with the design of a parallel parser. We showed that a parallel parser should allow each unification operation to be distributed independently and that it should implement a greedy or rule-based grouping heuristic. In addition, to allow the parser to make use of further developments and improvements on sequential parsers, we aim at keeping the design of the parallel parser as close to that of a sequential parser as possible.

The design of the parallel parser can best be explained by a step by step description of the parsing process. Each processor runs a thread that closely resembles a sequential parser. Each has its own agenda and chart. Initially, work associated with the input sentence is distributed amongst the agendas of the threads. Each thread continues processing the work on its agenda like a sequential parser, only inspecting its own chart and agenda. To allow two items owned by different threads to match, each thread inspects the items derived by other threads putting possible matches on its agenda. If the agenda contains work after this communication phase, it will continue processing as a sequential parser. Basically, all threads alter between sequential and communication mode until the entire parse has completed.

The threads incorporate synchronization mechanisms to detect overall termination and prevent duplicate matches of items. To allow each unification task to be distributed independently, threads are allowed to steal work from another thread by taking it from its agenda. The greedy heuristic for LinGO is implicitly implemented by using the distributed chart. The implementation for Deltra uses the rule-based heuristic.

The performance of the resulting parallel parser was evaluated by letting it run on a subset of the fuse test suite of sufficiently complex sentences to get accurate measurements. Running the parser on a SUN Ultra Enterprise 10000 with 64 200MHz UltraSparc processors yielded a maximum speedup of 31.4 at 48 processors. An overall peak was reached at 32 processors with an average speedup of 17.3. A drop in speedup for larger numbers of processors was caused by overhead in the scheduling algorithm. Running the full test suite on a SUN Ultra Enterprise with 8 400MHz processors yielded a speedup of 4.73 at 6 processors.

The memory bus often forms a performance bottleneck for multiprocessing. Efficient usage of cache can therefore be an important factor in obtaining speedup. One technique to improve cache utilization is to improve the spatial reuse of the unification algorithm. Memory is typically loaded in larger chunks. Feature structures typically span several of such chunks, whereas the chunks themselves can contain multiple nodes. When unification fails, usually only a fraction of the nodes is referenced. The idea is to store nodes of the feature structures in a depth-first order, the order in which they are typically accessed, in the hope to prevent loading as many chunks as possible in case of failure. The resulting algorithm was able to reduce the number of interference misses by 10%.

Cache utilization can also be improved by exploiting the freedom in order of evaluation of tasks on the agenda. We compared a FIFO and stack-based strategy against two blocking strategies. With blocking, operations are grouped on referencing a consecutive block of memory that just fits in the cache. This prevents cache from being repeatedly flushed when repeatedly iterating over a large set of data. The first blocking strategy, simple blocking, simply divides the chart in cache sized blocks. The second blocking strategy, categorized blocking, also groupes operations based on grammar rules. Experiments with Deltra showed that all blocking strategies performed better than a FIFO strategy. A stack-based strategy often performed better than simple blocking, though. One problem with simple blocking is that it often occurred that there were only two operations per block. Categorized blocking performed considerably better, with over 50% reductions of the number of interference misses compared to a FIFO strategy. The grouping of related operations also resulted in related items being stored in the same block. Using categorizing without blocking proved

to be less effective.

Conclusions

We have shown it is possible to considerably speed up the parsing of natural language with the use of shared-memory multiprocessing capabilities. By a thorough analysis of the parsing process, we were able to identify several limitations for parallel processing. The presented technique is likely to be useful for other natural language parsers, using different grammar formalisms, as well. Since the presented technique distributes work at the finest possible grain of distribution (apart from parallel unification), it is likely to achieve speedup as long as the average available parallelism is large enough. Investigating the possibilities of parallelism for such grammars can be done analogous to the techniques presented above.

Summary in Dutch

Parallel Ontleden van Natuurlijke Taal: Van Analyse tot Versnelling

Doel van Onderzoek

Onderzoek naar natuurlijke taalverwerking richt zich er op om computers de mogelijkheid te geven geschreven of gesproken natuurlijke taal te verwerken of te begrijpen. Voorbeelden van toepassingen zijn automatisch vertalen, natuurlijke taal interfaces, informatiewinning en het automatisch vinden van informatie. Toepassingen die proberen tot een volledig begrip van een gegeven tekst te komen gebruiken vaak ontleders die gebaseerd zijn op unificatiegrammatica's. Kortweg kunnen dit soort ontleders beschouwd worden als contextvrije ontleders waaraan extra restricties worden toegevoegd in de vorm van unificatie van grafen.[1] Dit soort ontleders nemen vaak het grootste deel van de verwerkingstijd van een zin voor hun rekening. Daarvan is vaak meer dan 90% nodig voor het uitrekenen van de unificaties.

De meeste applicaties voor natuurlijke taalverwerking zijn ontworpen voor verwerking met slechts één processor. De steeds populairder wordende shared-memory multiprocessoren lijken een goede mogelijkheid te bieden om taalverwerking met enkele factoren te versnellen. In gevallen waar een applicatie meerdere zinnen onafhankelijk van elkaar kan verwerken, is het effectief benutten van meerdere processoren triviaal. Bij ons onderzoek gaat het er dan ook voornamelijk om het ontledingsproces zelf te parallelliseren. Dit is vaak de enige oplossing voor toepassingen waar het verwerken van een enkele zin bepalend is voor de verwerkingstijd. Voorbeelden van dit soort toepassingen zijn toepassingen die directe interactie met de gebruiker vereisen.

Een van de meest populaire ontledingstechnieken voor natuurlijke taalverwerking is chart ontleding. Bij deze techniek wordt een chart gebruikt om tussentijdse resultaten op te slaan. Deze tussentijdse resultaten worden gebruikt om uiteindelijke resultaten op een efficiënte manier op te slaan en om het dupliceren van werk te voorkomen. Chart ontleders hebben vaak ook een agenda die een lijst bijhoudt van taken die nog door de ontleder moeten worden uitgevoerd. In ons onderzoek hebben wij chart ontleders als uitgangspunt gebruikt voor parallellisering.

Het onderzoek richtte zich in eerste instantie op het ontwikkelen van een parallelle ontleder voor de Deltra grammatica (ontwikkeld op de TU Delft). Later werd er ook een ontleder ontwikkeld voor de LinGO grammatica van CSLI Stanford. Aangezien de mogelijkheid om versnelling te behalen voor een groot deel wordt bepaald door de grammatica, is het wenselijk om de ontwikkelde technieken voor meerdere grammatica's te verifiëren. Een ander voordeel van het betrekken van LinGO bij het onderzoek is dat LinGO geleverd wordt

[1]Het moet echter opgemerkt worden dat er tegenwoordig aan het ontwerp van grammatica's vaak niet een expliciete contextvrije "backbone" ten grondslag ligt.

met een uitgebreide set van test bestanden en gereedschappen om de prestaties van verschillende ontleders met elkaar te vergelijken.

De Haalbaarheid van Parallel Ontleden

Het implementeren van een multi-threaded toepassing kan erg moeizaam zijn. Gelukkig is het mogelijk om uit de oorspronkelijke sequentiële ontleders nuttige informatie af te leiden die het maken van ontwerpkeuzes kunnen vergemakkelijken. Als eerste hebben we gekeken naar de hoeveelheid parallellisme die in het ontleedproces aanwezig is. Het is bekend dat de verwerkingstijd T_P van elke multi-threaded toepassing bij een gebruik van P processoren beperkt kan worden tot

$$T_P \leq T_1/P + T_\infty,$$

waar T_1 de verwerkingstijd is op één processor en T_∞ het kritieke pad, of de minimale verwerkingstijd. Tevens wordt de maximale versnelling die behaald kan worden beperkt door het gemiddeld beschikbare parallellisme T_1/T_P. De waarde T_∞ konden we afleiden door de verwerking van iedere zin als een taakgraaf te representeren. In deze graaf worden taken als knopen gerepresenteerd. Voor iedere taak wordt tevens de benodigde uitvoeringstijd vastgelegd. Takken geven de afhankelijkheden tussen de knopen aan. Het langste pad in een taakgraaf komt overeen met T_∞.

We hebben in ons onderzoek twee verschillende definities voor taakgrafen bekeken. Voor het eerste type taakgraaf hebben we een taak gelijkgesteld aan één unificatie operatie. Aangezien het bekend is dat unificatie moeilijk te parallelliseren is, is het gebruikelijk om unificatie als een ondeelbare operatie te zien. Voor het tweede type taakgraaf hebben we een taak gelijkgesteld aan een aantal gerelateerde operaties.

Uit de experimenten bleek dat alleen als iedere unificatie als een aparte taak werd uitgevoerd, het gemiddeld beschikbare parallellisme voldoende was om versnelling te kunnen verwachten. Dit impliceert dat een parallelle ontleder alleen versnelling kan halen als het toelaat iedere unificatie afzonderlijk te verdelen tussen de processoren.

Een ander belangrijk aspect dat de mogelijkheden van parallel ontleden bepaalt, is de benodigde communicatie tussen de processoren. Ondanks dat alle processoren van een shared-memory systeem direct toegang tot al het geheugen hebben, speelt communicatie een belangrijke rol in de gedaante van cache misses. Om de complexiteit van de simulaties te beperken hebben we echter een netwerk model gebruikt om communicatie te analyseren. Uitgaande van de vorige resultaten, zijn we ervan uitgegaan dat iedere unificatie als een aparte taak moet worden beschouwd.

Als eerste resultaat vonden we dat het willekeurig distribueren van taken meestal tot zeer grote hoeveelheden communicatie leidt. Een oplossing voor dit probleem is om taken die dezelfde data gebruiken op dezelfde processor te groeperen. Het is niet haalbaar om tijdens het ontleden een optimale verdeling van taken te vinden. Daarom zijn we aangewezen op het gebruik van heuristieken. Ook mogen de heuristieken het verdelen van individuele unificaties niet belemmeren. Een goede heuristiek zal de communicatie beperken terwijl het gelijktijdig een goed gebalanceerde verdeling van werk toelaat.

We hebben drie verschillende groeperingsheuristieken geëvalueerd. De heuristieken groeperen op respectievelijk het deel van de zin die door een taak wordt bestreken, de grammaticaregel die een taak toepast en een greedy aanpak waarbij een taak wordt uitgevoerd op de processor die de benodigde data heeft opgeslagen.

Opnieuw werden de metingen per zin uit de fuse test set uitgevoerd. Voor iedere heuristiek bepaalde we vervolgens een willekeurige distributie van de taken (na groepering) en een benadering van het beste geval. De greedy aanpak leverde de beste resultaten voor de LinGO grammatica. Het leverde de meest gebalanceerde distributie, wist de bovenste limiet van de communicatie goed te beperken en liet bovendien de mogelijkheid open om dicht bij het absolute minimum aan communicatie te komen. Voor Deltra bleek de per regel groepering goede resultaten te leveren. Deltra heeft meer grammatica regels dan LinGO en gebruikt bovendien een andere ontledingsstrategie waarvoor de per regel groepering zeer geschikt is.

De Implementatie van een Parallelle Ontleder

De eerste stap in de ontwikkeling van een parallelle ontleder is het vinden van een thread-safe unificatiealgoritme. Zonder zo'n algoritme is het niet mogelijk om dezelfde graaf in verschillende unificaties tegelijk te gebruiken. Deze beperking kan de ontwikkeling van een parallelle ontleder aanzienlijk bemoeilijken, vooral als "structure sharing" wordt gebruikt (wat meestal het geval is).

Ons thread-safe unificatiealgoritme is gebaseerd op Tomabechi's algoritme. Tomabechi's algoritme wordt vaak gebruikt omdat het bekend staat als een van de snelste unificatiealgoritmes voor natuurlijke taalverwerking. Deze snelheid wordt behaald door het gebruik van administratieve velden, of kladvelden, die zijn opgenomen in de knopen van de grafen. Afgezien dat het gebruik van deze velden ervoor zorgdragen dat het algoritme niet thread-safe is, nemen deze velden ook nog eens onnodig geheugen in beslag als de gerelateerde knoop niet in een unificatie betrokken is. Ook andere populaire unificatiealgoritmes gebruiken vaak kladvelden.

Het idee achter het thread-safe maken van deze unificatiealgoritmes is om de kladvelden uit de knopen te halen en deze tijdens uitvoering van de unificatie met de knopen te associëren. Op deze manier kunnen verschillende threads tegelijk ieder een aparte buffer met kladvelden met dezelfde knoop associëren. Grafen kunnen dan ongestraft in meerdere unificaties tegelijk voorkomen. Bovendien hoeven kladvelden alleen maar tijdens de unificatie aan een knoop te worden toegekend. De kladvelden kunnen dus na iedere unificatie opnieuw worden gebruikt, waardoor het totale geheugengebruik afneemt.

We hebben twee verschillende manieren onderzocht om klad buffers met knopen te associëren. Bij de eerste aanpak wordt hashing gebruikt om het fysieke adres van een knoop te koppelen aan een buffer. Bij de andere aanpak worden knopen een unieke index in een array van kladbuffers toegekend. Bij iedere graaf worden knopen in een "depth-first" volgorde genummerd. Aangezien grafen indexen gemeen zullen hebben, wordt tijdens unificatie aan iedere graaf bovendien een offset toegekend die bij deze index moet worden opgeteld.

Ook al vergt het indexeren van knopen extra werk, is het laatstgenoemde algoritme toch ongeveer even snel dan Tomabechi's algoritme. Dit kan verklaard worden door het feit dat

de indexeringsmethode beter gebruik maakt van de cache doordat de kladvelden worden hergebruikt. Het indexeren bleek ook ongeveer 10% sneller te zijn dan hashing. Zowel indexeren en hashing leverde een reductie in geheugengebruik op van ongeveer 50% voor LinGO en 75% voor Deltra.

Nu we een thread-safe unificatiealgoritme hebben, kunnen we verder met het ontwerpen van een parallelle ontleder. We hebben laten zien dat een parallelle ontleder moet toestaan dat de unificaties onafhankelijk van elkaar gedistribueerd moeten kunnen worden en dat het een greedy of regel georiënteerde groeperingsheuristiek moet gebruiken. Om verdere ontwikkelingen aan sequentiële ontleders makkelijk op te kunnen nemen in ons ontwerp, is het wenselijk dat het ontwerp zoveel mogelijk aansluit bij dat van een sequentiële ontleder.

Het ontwerp van de parallelle ontleder kan het best worden uitgelegd aan de hand van een stap voor stap beschrijving van het ontledingsproces. Op elke processor wordt een thread gestart die in grote lijnen dezelfde werking heeft als een sequentiële ontleder. Iedere thread heeft zijn eigen chart en agenda. Bij het opstarten wordt het initiële werk dat overeenkomt met de gegeven zin verdeelt over de agenda's van de threads. Vervolgens verwerkt iedere thread zijn agenda als ware het een sequentiële ontleder. Om toch de tussentijdse resultaten van de verschillende threads te kunnen combineren, inspecteert vervolgens iedere thread de resultaten van de andere threads. Nieuw werk wordt op de agenda geplaatst. Als een thread inderdaad nieuw werk heeft gevonden hervat het zijn rol als sequentiële ontleder. Threads blijven wisselen tussen de sequentiële en communicatie mode totdat al het werk gedaan is.

Het algoritme dat iedere thread draait bevat synchronisatiemechanismes om terminatie te detecteren en het dubbel afleiden van dezelfde resultaten te voorkomen. Om het mogelijk te maken iedere unificatie afzonderlijk te verdelen, is het een thread mogelijk gemaakt werk van de agenda's van andere threads te stelen zodra de thread in kwestie zonder werk komt te zitten. De greedy groeperingsheuristiek is impliciet geïmplementeerd door het gebruik van de gedistribueerde chart.

Om de prestaties van de resulterende parallelle ontleder te testen is een subset van de fuse test set genomen van alle zinnen die complex genoeg zijn om accurate metingen te verkrijgen. Op een SUN Ultra Enterprise 10000 met 64 200MHz UltraSparc processoren leverde de ontleder een maximale versnelling van 31.4 bij 48 processoren. Een gemiddelde top werd bereikt bij 32 processoren met een gemiddelde van 17.3. Door overhead in het verdelingsalgoritme werd voor hogere aantallen processoren een lagere versnelling gehaald. Het uitvoeren van de volledige fuse test suite op een SUN Ultra Enterprise met 8 400MHz processoren leverde een versnelling van 4.73 bij 6 processoren.

De geheugenbus is vaak een zwakke schakel bij multiprocessing. Goed gebruik van de cache kan daarom van groot belang zijn om de gewenste versnelling te halen. Een manier om het cachegebruik te verbeteren is om het spatiële hergebruik van het unificatiealgoritme te verbeteren. Geheugen wordt normaliter in grotere blokken tegelijk in de cache geladen. Grafen nemen vaak meerdere van deze blokken in beslag, terwijl er juist meerdere knopen in een blok passen. Als unificatie faalt hoeft er vaak maar een fractie van de knopen te worden ingeladen. Het idee is om knopen in dezelfde volgorde op te slaan als ze doorgaans worden ingeladen, zodat de in te laden knopen zoveel mogelijk in dezelfde blokken gegroepeerd zijn. Het resulterende algoritme leverde een 10% reductie van het aantal interference misses op.

Het cache gebruik kan ook verbeterd worden door gebruik te maken van de vrijheid in de volgorde van evaluatie van de taken op de agenda. We vergeleken een FIFO en LIFO strategie met twee blockingstrategiën. Bij blocking, worden operaties gegroepeerd die hetzelfde aaneensluitende in cache passende blok geheugen aanspreken. Dit voorkomt dat de cache steeds wordt gewist als er herhaaldelijk wordt geïtereerd over hetzelfde grote stuk geheugen. Bij de eerste blockingstrategie, simpele blocking, wordt de chart simpelweg in blokken verdeeld ten grootte van de cache. Bij de tweede strategie, gecategoriseerde blocking, worden operaties daarbovenop ook nog per grammaticaregel gegroepeerd. Bij experimenten met Deltra bleek blocking het altijd beter te doen dan een FIFO strategie. Een LIFO strategie deed het echter vaak beter dan simpele blocking. Een van de problemen die voorkwam bij simpele blocking is dat er vaak maar twee operaties per block per keer werden uitgevoerd. Voor gecategoriseerde blocking waren de resultaten aanzienlijk beter. Deze leverde een meer dan 50% reductie van het aantal interference misses op ten opzichte van de FIFO strategie. Het groeperen van gerelateerde operaties had tot gevolg dat gerelateerde data ook in dezelfde blokken werd opgeslagen. Het gebruik van enkel categorisering zonder blocking leverde minder goede resultaten op.

Conclusies

We hebben laten zien dat het mogelijk is om met behulp van shared-memory systemen aanzienlijke versnellingen te behalen bij het ontleden van natuurlijke taal. Door eerst het ontledingsproces grondig te analyseren, konden we in een vroegtijdig stadium de beperkingen van parallel ontleden identificeren en daar het ontwerp op afstemmen. De resulterende techniek voor parallellisatie is zeer waarschijnlijk ook goed te gebruiken voor andere ontleders van natuurlijke taal, zelfs als die andere grammatica's gebruiken. Aangezien onze ontleder werk op de meest fijnkorrelige manier distribueert (afgezien van parallelle unificatie), is het waarschijnlijk dat het mogelijk is met deze techniek versnelling te halen als inderdaad de hoeveelheid parallellisme toereikend is. Een voortijdige analyse, analoog aan degene waar wij onze ontleders aan hebben onderworpen, kan uitmaken of het behalen van versnelling in deze gevallen tot de mogelijkheden behoort.

About the Author

The author of this thesis was born on October 28, 1973, in Alphen aan den Rijn, The Netherlands. In 1992, after receiving his high-school diploma at the atheneum level from the "Ashram College", he stared a study in Computer Science at the "Delft University of Technology". In 1997 he received his M.Sc. degree cum laude, after which he continued at this university as a Ph.D. student. The topic of his Ph.D. research was "Parallel Natural Lanaguage Interfaces". Obviously, this thesis is a result of this research. He is currently employed at "YY Technologies", in Mountain View, California.

Index

a
a-way associative, 43, 107
abstract typed feature structure, 19
active chart, 29
active edge, 12
active item, 12, 29, 47
affinity scheduling, 39
agenda, 29, 38, 90, 100, 114
appropriate features, 20
attribute value matrix, 18
average available parallelism, 37, 50, 54, 60, 72, 118, 132

b
balance, 42, 65
blocking, 45, 112, 113, 134
bottom-up, 12, 29, 56, 59

c
cache line, 43, 72, 105
cache miss, 44, 105
CaLi, 5
capacity misses, 44, 45, 106, 107, 113, 115
categorized blocking, 113, 135
centralized agenda, 31, 32, 33, 90, 119
centralized chart, 31, 33, 87
chart parser, 29
chart, 29, 80, 90
Chomsky Normal Form, 8, 12
coherency misses, 44, 65
communication graph, 64, 65, 69
communication volume, 41
complex node, 18
compulsory misses, 44, 106–108, 113
concurrent unification, 28, 37, 47, 71, 72, 74, 85, 87, 88, 90
conflict misses, 44, 108
consistent, 19
constraint function, 19
constraint, 20, 24, 77
context-free grammar, 8
critical path overhead, 40
critical path, 38, 47, 50

d
dag depth, 38, 51
deduction rules, 10, 48, 66
deduction sequence, 11
Deltra, 4, 5
derivation tree depth, 56, 100
derived items, 48, 58
destructive, 23
direct mapping, 43
distributed agenda, 31, 33, 90, 119
distributed chart, 31, 67, 87, 88, 134
domain decomposition, 41
dynamic filtering, 11
dynamic scheduling, 39

e
edge cut, 41, 64
edge list, 90, 94
efficiency, 37
epsilon rules, 8, 14, 16, 48, 49
exhaustive parsing, 29, 118, 119

f
false sharing, 44
filtering, 31, 33, 48, 58, 69
footprint, 107
foreign edges, 90, 92, 117
fully associative cache, 43, 45

g
generation, 24
grammar rules, 21, 31, 133
greedy, 67, 87–89
greedy schedules, 38, 40, 98, 99
grouping heuristic, 65, 87, 89, 113, 132, 134
grouping, 65

guided self-scheduling, 39

h
Head-Driven Phrase Structure Grammar, see HPSG 18
HPSG, 18, 118
hypotheses, 10, 48

i
incremental copying, 24
inference relation, 11
initialization items, 48, 51
input string, 8, 48, 67
instance, 20, 31
interference misses, 44, 46, 107, 113, 134
item domain, 10, 14

j
justification graph, 48, 50

k
key-driven, 15

l
language, 8
lazy copying, 24
least recently used, see LRU 44
look-ahead, 11
length, 51
level, 43
Lexical-Functional Grammar, see LFG 18
LFG, 18, 118
LinGO, 4, 118
LKB, 5
local edges, 90
locality grouping, 45, 113, 114
LRU, 44, 45, 114

m
match tasks, 51, 63, 69
match, 29, 48, 52, 90, 92, 94
maximum total communication, 64, 65, 69, 89
memory line, 43, 107

n
n-first parsing, 29, 119
nominal speedup, 37, 40
non-terminal, 8, 66

p
parallel complexity, 37
parallel tabular chart parser, 31, 32, 34, 39, 67
parallel unification, 28, 47, 72, 85, 118
parser, 8
parsing schema, 10, 11, 47, 66, 67, 88
parsing system, 10, 47, 95
partitioning, 41
passive edge, 12
passive item, 12, 21, 29, 47, 80
path, 18, 31, 48
path equivalence, 19
path value, 19
precede, 48
production rules, 8, 66

q
quick check, 31, 34, 52, 96, 118

r
reading, 9
recognizer, 8
recognizes, 8, 10
recursive spectral bisection, see RSB 42
reentrancy, 19, 77, 79, 111
reentrant, 19
representative, 22
reuse, 44, 72, 112
roots, 21
RSB, 42
rule filter, 31, 96
rule-based, 66, 87–89, 113

s
schedule, 37, 38
scheduling, 36
scratch buffer, 72, 133
scratch fields, 71, 72, 83, 85, 90, 105, 133
self scheduling, 39
sequential complexity, 37
set-associative mapping, 43, 114
shared, 43

simple blocking, 112, 135
spatial reuse, 44, 45, 105, 111
speedup, 37, 41
start symbol, 8, 21
static filtering, 11
static scheduling, 39
stealing policy, 40
step contraction, 11, 16
strict, 49
string chart parser, 31
structure sharing, 26, 35, 64, 133
subsumes, 20

t
tabular chart cell-based, 67, 113
tabular chart, 30, 31, 117
tabular parsing, 28
task, 50
task dependency graph, 50, 53, 55, 58, 63, 99
task graph, 36, 53
temporal reuse, 44, 45, 112
terminal, 8
thread grouping, 53
tiling, 45
top-down prediction, 12, 29, 32, 56
transition function, 18
type 1, 54, 58
type 2, 54, 55
type hierarchy, 19, 80, 85
type system, 19
typed feature structure, 18

u
unification, 20
uniform-sized chunking, 39
unify–verify–match cycle, 30, 32, 52–54, 88, 94
uninstantiated parsing system, 10, 47

v
verification, 29, 85, 87

w
well-formed, 20, 27
work, 38, 50
work overhead, 40
work sharing, 39
work stealing, 39, 89, 93
work-first principle, 40, 60, 94