Effective Exploitation of Parallelism in NLP

M. P. van Lohuizen

Dept. of Information Technology and Systems
Delft University of Technology
Zuidplantsoen 4, 2628 BZ Delft, The Netherlands
mpvl@acm.org

Keywords: Natural Language Processing, Parallel Parsing, Fine-Grained Parallelism

Abstract

The increasing demand for accuracy and robustness for today's natural language processing systems brings on an increasing demand for computing power. Experiments on our unification-based parser indicated that only fine-grained parallelism could yield acceptable speedup. We present a parallelization technique to exploit this fine-grained parallelism. Although the approach was designed for the parallelization of a unification-based parser, most of the complexity is captured in an independent module that can be used for other applications as well.

1 Introduction

The performance of natural language processing (NLP) applications has always been limited by the speed of the workstations at hand. Whenever faster computers entered the market, NLP applications were extended to deal with even more complex linguistic phenomena and an even larger domain of discourse. With the introduction of robust parsing techniques (parsing of partially correct sentences), computational demands have increased even further. The increasing availability and affordability of multiprocessor machines makes the exploitation of parallelism that is inherent in NLP applications an important mean to improve the performance even further.

In this paper, we will present techniques to parallelize unification-based parsers. Unification-based parsers are one of the most common components in NLP systems. They are mostly used for syntactical analysis, but can also be used for, e.g., semantical analysis. Unification-based parsers are often also the most computationally expensive component. Together, this makes them ideal candidates to investigate the possibilities for exploitation of parallelism.

A unification-based parser is basically a context-free parser augmented with unification, which can be seen as an extra constraint on the context-free backbone. Often, the unification operations (which are similar to those found in Prolog) are the most costly part of parsing. Basically, one can take two approaches to parallelizing a unification-based parser. First, one can take unification (and subsumption) as an atomic operation, and exploit the parallelism contained in the context-free part of parsing. Second, one could parallelize the unification operation itself. Most research has focussed on the former approach.

One of the reasons for this choice is that unification is not in Nick's Class and therefore generally considered to be a hard to parallelize problem [1, 2]. In [3], however, it is shown that this property does not mean it is not feasible to make an efficient parallel unification algorithm (see also remarks in [4]). There are, however, other reason that make the former approach more attractive. The parallelism inherent in the unification algorithm is very fine-grained. This ensures that work can be balanced evenly amongst processors, but also increases the potential overhead of the system. In addition, this fine granularity makes it hard to implement on distributed systems. Nevertheless, despite all arguments in favor of the first approach, most attempts to exploit parallelism at the context-free level were not very successful (e.g. [5, 6], [7], [8]).

Experiments in [9] suggest that many of the failed attempts to come to an efficient parallel parser can be attributed to the nature of the parallelism inherent in unification-based parsers. In this paper, we will present some of these experimental results. After pinpointing the problems, we will present an alternative parallelization strategy, and present a parallel unification-based parser. In the remainder of the paper we will evaluate the results.

2 Related Work

For over a dozen of years, attempts have been made to exploit the parallelism that is inherent in NLP tasks. Approaches ranged from distributing functionally different components amongst different processors, to exploiting the parallelism within a single component of NLP applications (see [13] for an overview).

Parallel parsing for NLP in specific has been researched extensively. For example, [14] presented a parallel LR parser, and [5, 6] presented some implementations of parallel chart parsers. In [15] we can find a more theoretical overview of parallel chart parsers. There has also been research focusing on the parallelization of alternatives for conventional parsing: for example, in [16] we can find a parallel memory-based parser, and [17] discusses an alternative to parsing geared towards parallel processing.

Whereas most research focussed on parallelizing at the context-free level, in [18], [4], and [3] we can find research on parallel unification. This research is complementary to the other research, because it focuses on a completely different part of parsing.

Recently, several parallel NLP systems appeared. In [19], two existing NLP applications were successfully parallelized using the parallel Prolog environment ACE. Even though a great deal of NLP applications are written in Prolog, the ones that are not cannot benefit from this approach. Our approach can bring a solution to this category of systems, and hence is complementary to the research in [19]. We even suspect that our parallelization technique can be used to parallelize tabular Prologs (like XSB and Dialog).

In [20], a parallel parser generator based on the Eu-PAGE system is proposed. This system builds on top of Orchid and PVM, using a distributed shared-memory model. In addition to being very specific to parsing, this solution exploits coarse-grained parallelism of the kind that proved unusable for the Deltra system. We suspect that many other NLP systems cannot benefit from this approach either (see, e.g., remarks in [7]). Our approach provides a solution in case fine-grained parallelism is required.

3 Analysis of Parallelism

The Deltra system was used as the basis of our experiments. It is based on a DCG-like formalism, as found in Prolog. The grammar used in the experiments covers a wide range of linguistic phenomena of the Dutch language. It is specified as a separate syntactic grammar (± 360 ECFG-based rules, yielding $\pm 15,000$ generated generated CFG-based rules), morphological grammar (± 120 rules), and dictionary (± 3000 entries). The syntactical component also includes a minimal amount of semantic specifications.

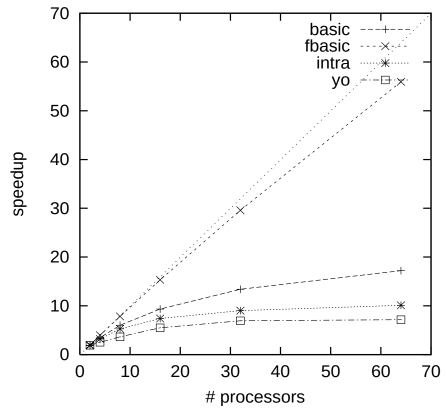


Figure 1: Upper bound of speedup for different distribution techniques

A tabular-chart, double-dotted parser is used to parse the input ([10]).

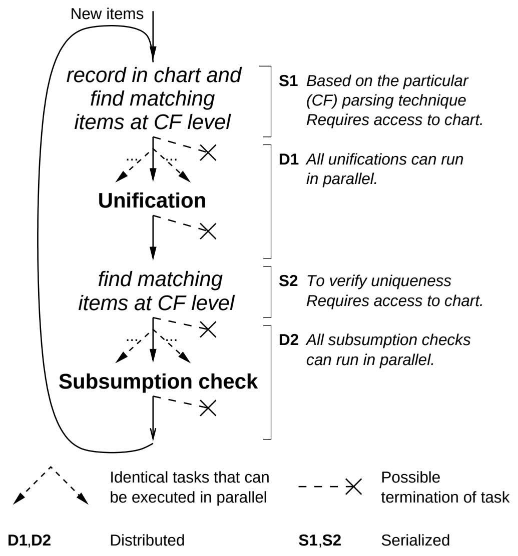
In [9], we analyzed several techniques to parallelize unification-based parsers, varying in the way work and items are distributed among processors. The experiments were aimed at finding an upper bound for the speedup for these different techniques. Figure 1 shows the results for the four best techniques. As suspected, most upper bounds indicated poor performance, meaning that regardless of the implementation, such techniques could never obtain acceptable performance.[1]

These results can be explained as follows. Because of memory limitations, most distribution schemes aimed at distributed architectures (as opposed to shared-memory systems) ultimately store each item on only one processor. Henceforth, all operations involving this item will be performed by the respective processor. However, from analyzing many of Deltra's parsings, we know that some items are often needed in parsing operations an order of thousand times more than average, hence making the schemes vulnerable to unbalanced distribution.

A similar problem occurred for the commonly used basic chart approach. With the basic chart approach, each time an unprocessed item is picked up from the agenda, all processing for this item is performed by a single processor. This resulted in unbalanced distribution of work, because the amount of work associated with items varied wildly.

The fine-grained basic chart distribution allows each unification and subsumption operation to be performed by a different processor. It appeared to be the only distribution scheme with the potential to achieve near-linear speedup. The differences between the sizes of the feature structures did not thwart perfor-

[1]It is not necessary, within this context, to discuss the details of the actual distribution techniques.



New items
record in chart and find matching items at CF level
S1 Based on the particular (CF) parsing technique Requires access to chart.
D1 All unifications can run in parallel.
Unification
find matching items at CF level
S2 To verify uniqueness Requires access to chart.
D2 All subsumption checks can run in parallel.
Subsumption check
Identical tasks that can be executed in parallel
Possible termination of task
D1,D2 Distributed
S1,S2 Serialized

Figure 2: Control flow of fine-grained basic chart.

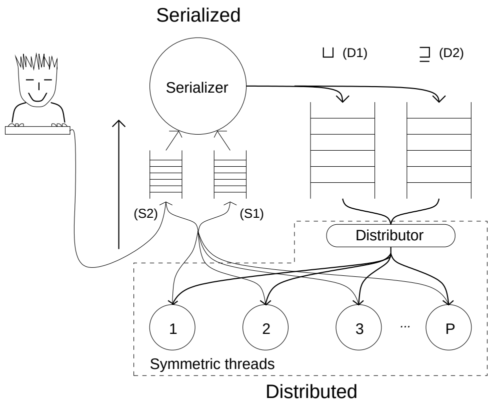


Figure 3: System architecture.

mance, although it does explain the slight deviation from optimal speedup for a larger amount of processors.

An outline of the scheme, which is intended for shared-memory systems, is given in figure 2. All threads executing blocks marked D1 or D2 (**D-blocks**) can run in parallel. The S1- and S2-blocks (**S-blocks**) are not thread-safe, because each block requires exclusive access to the chart. (The chart is used to prevent doing redundant work.) We call a single execution of a D-block a **D-task** and a single execution of an S-block an **S-task**.

4 Exploiting Fine-Grained Parallelism

The most important guideline for parallelizing on a shared memory system is to reduce idle time of processors as much as possible. From Graham's findings we know that if we create a new thread at the start of each D-block (see figure 2), and lock each S-block, it will be possible to achieve at least half the optimal speedup [11]. However, as we have argued, the only way to achieve an acceptable speedup is to distribute each individual unification and subsumption. For typical NLP systems, this can amount to hundreds of thousands of threads per sentence, most likely yielding many seconds on task creation alone.

To prevent any overhead from thread creation at all, we have chosen to assign a single thread to each processor. This brings in the problem that simply locking the S1- and S2-blocks could idle processors for a considerable period of time. Although no guarantee for success, we could also devise a more fine-grained locking scheme. This, however, is a laborious error-prone task and is likely to increase the overhead.

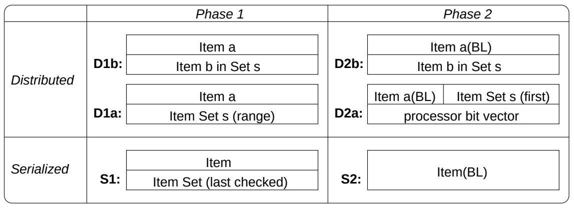
4.1 Minimum-Overhead Scheduling

A radically different approach is based on the observation that not all four blocks (S1, D1, S2, D2) need to be executed consecutively by a single processor. Instead, we enable each block to be scheduled independently. The resulting system is outlined in figure 3. The **serializer** is responsible for processing S-tasks. The **distributor** merely passes D-tasks to the symmetric threads. Because S-tasks will only be executed by one processor, they need not be surrounded by locks. Conversely, a symmetric thread need not wait for locks at the start of an S-block; it will only need to briefly lock to queue up the respective S-task.[2]

Obviously, this scheme is only useful if the critical region for queueing an S-task is much smaller than taking the S-block itself as a critical region. In addition, in order to obtain a speedup of n, the total amount of serialized work should not exceed $1/n$ of the total amount of work. Indeed, it holds that the S-tasks are much less computationally expensive than D-tasks; yet they can be quite expensive, because they manipulate complex indexing structures, among other things. Moreover, in our case, there are many more D-tasks than S-tasks, because many D-tasks will fail without ever queueing an S-task.

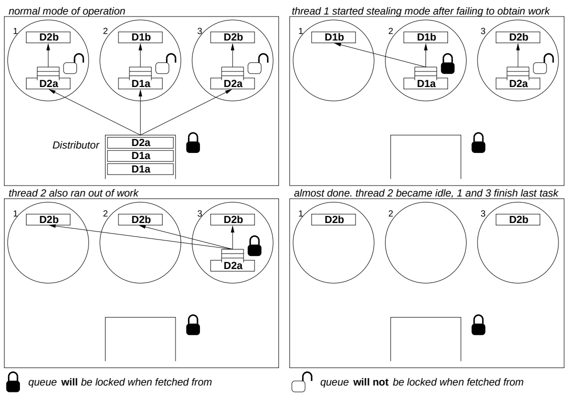
Because we have split the loop into separately schedulable parts, we need a way to encode the state of processing at the end of a block, so that any thread can continue processing the succeeding block at any time. We will call such encodings **task descriptors**. Figure 4 gives a schematic overview of the different task descriptors. (The details will be discussed in the next section.) We will denote a task descriptor and the associated work as T, where a subscript is used to indicate its type. As we can see, all D-tasks are encoded by two task descriptors ($T_{\mathrm{D}n\mathrm{a}}$ and $T_{\mathrm{D}n\mathrm{b}}$). The reason for this is as follows. An S-task can spawn many

[2]It is not hard to eliminate locking altogether. Due to additional overhead, though, such solutions are often not faster.



		Phase 1		Phase 2
Distributed	**D1b:**	Item a / Item b in Set s	**D2b:**	Item a(BL) / Item b in Set s
	D1a:	Item a / Item Set s (range)	**D2a:**	Item a(BL), Item Set s (first) / processor bit vector
Serialized	**S1:**	Item / Item Set (last checked)	**S2:**	Item(BL)

Figure 4: Task descriptors



normal mode of operation
thread 1 started stealing mode after failing to obtain work
thread 2 also ran out of work
almost done. thread 2 became idle, 1 and 3 finish last task
D1b
D2b
D1a
D2a
Distributor
queue will be locked when fetched from
queue will not be locked when fetched from

Figure 5: Stealing of symmetric threads.

D-tasks (see figure 2). Alternatively, an S-task can also be seen as spawning a single task performing a series of unification (or subsumption) operations for a single item against a set of other items. We use task descriptors of type $T_{\mathrm{D}x\mathrm{a}}$ to specify the latter type of task, whereas we use $T_{\mathrm{D}x\mathrm{b}}$ to represent a single unification or subsumption operation. This two-level distinction allows us to let S-tasks put a single $T_{\mathrm{D}n\mathrm{a}}$ on the queue, and delay the expansion of $T_{\mathrm{D}n\mathrm{a}}$ into instances of $T_{\mathrm{D}n\mathrm{b}}$ until these are actually up for execution. Note that it is straightforward to make the expansion $T_{\mathrm{D}n\mathrm{a}}$ descriptors into $T_{\mathrm{D}n\mathrm{b}}$ descriptors thread-safe.

There are several advantages to this scheme. For one, the sizes of the distributed queues are reduced considerably, because we do not need to create task descriptors of type $T_{\mathrm{D}x\mathrm{b}}$ until they are needed. More importantly, though, it also allows for yet another optimization. Our experiments showed that many unification and subsumption operations fail almost instantly. In these cases, the overhead of locking the distributor can become considerable. To avoid this overhead, we can simply delegate the processing of a $T_{\mathrm{D}x\mathrm{a}}$ to one of the symmetric threads. To such a thread, $T_{\mathrm{D}x\mathrm{a}}$ descriptors would simply be queues of $T_{\mathrm{D}x\mathrm{b}}$ tasks. This, however, increases the grain of the parallelism, because now an entire $T_{\mathrm{D}x\mathrm{a}}$ task is processed by a single thread. What we would like to have is some mechanism that automatically changes the grain of parallelism as required. These facilities are provided by the scheduler described in [12], on which our parallel parser is based.

In the resulting scheme, symmetric threads normally only fetch work from the distributor ($T_{\mathrm{D}x\mathrm{a}}$ tasks). However, whenever the distributor is out of work, threads are allowed to steal work from other threads. This process is depicted in figure 5. The system enforces exclusive access to all $T_{\mathrm{D}x\mathrm{a}}$. The main advantage of using this mechanism is that, during normal operation, the overhead of performing a fine-grained $T_{\mathrm{D}x\mathrm{b}}$ task is negligible. Only when a thread starts stealing from another thread's $T_{\mathrm{D}x\mathrm{a}}$, both threads will start locking access to this $T_{\mathrm{D}x\mathrm{a}}$. There is no additional overhead during the transition from normal mode to stealing mode.

We conclude describing a final optimization. Instead of letting the serializer be a separate thread, we let it be a procedure that is called periodically by any of the symmetric threads. Symmetric threads attempt to gain exclusive access to this procedure each time they need to queue an S-task. If this fails, they simply queue the work. Otherwise, they process queued S-tasks and process their own S-task immediately, eliminating the need to queue the latter. So, queueing of work is used as a fall-back only, reducing the overhead even more.

All queueing and locking mechanisms, including the stealing mechanism, are implemented as an independent system described in [12]. It can be reused for all parallelization efforts compatible with this approach. In the next section we will elaborate on the details specific to parallel parsing.

4.2 Parallel Parsing

We will explain the implementation of the parser by walking through a single cycle of the parsing process (see figure 2). We will start describing the execution of a D1-task.

Basically, D1-tasks perform a single unification operation. We have chosen a non-destructive unification algorithm that makes use of binding lists, because this makes it straightforward to make unification thread safe. If one is willing to trade memory for efficiency, it is also possible to use a variant of Tomabechi's algorithm [4], where each node's scratch registers are duplicated for each processor. If unification succeeds at the end of a D1-task, the new item is used to represent an S2-task. Because we do not make a copy of the feature structure until after the uniqueness test has succeeded, we pass the binding list as well.

An S2-task basically does nothing more than looking up the item set in which the new item should be placed. If this set is empty, we can just add the item. Otherwise, the uniqueness of the new item needs to be verified by testing subsumption with existing items.

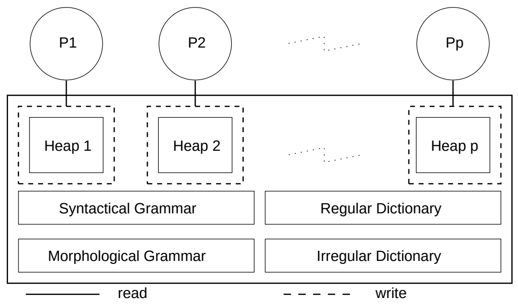


Figure 6: Layout of the different heaps.

This is resembled by the task descriptor T_{D2a}. It contains a reference to both the item and the item set.[3] Since we still do not copy the item, we pass the binding list as well. We also record the set's first item. The use for this will be explained later.

D2-tasks are very similar to D1-tasks, as unification is comparable to subsumption. After an T_{D2a} descriptor is obtained, it is expanded into a series of T_{D2b} descriptors, one by one. If a T_{D2b} succeeds (i.e., the item is not unique), the T_{D2a} is marked as done so that the rest of the T_{D2b} will not be executed. The scheme gets more complicated in case other processors steal from T_{D2a}, because we need to detect the last thread that fails. We let the last T_{D2b}-task to fail be responsible for creating the T_{S1} descriptor. Because we let each thread that is stealing from a T_{D1a} set its bit in the bit vector associated with T_{D1a}, it can detect whether it processed the last T_{D2b} by verifying that T_{D1a} is empty and all bits in the bit vector are zero. Because of the two-stage work distribution, this extra work is only required when threads are in stealing mode.

If all subsumptions fail, the feature structure is written to the heap before it is passed to the T_{S1} descriptor. This is a thread-safe operation, because we applied the heap layout as depicted in figure 6. Each processor has exclusive write access to its private heap. All processors have read access to all heaps. Since feature structures are not used in items before they are completed, there can be no conflicts. This scheme is also likely to improve caching behavior, because all writes of one processor are localized in one region.

D2-tasks also record the last item checked for subsumption in T_{S1} descriptors. This allows S1-tasks to verify all items have been checked for subsumption. If another item was added to the item set simultaneously, a new T_{D2a} descriptor will be created specifying the first item in the set at which checking should continue. If everything was checked, an S1-task creates a T_{D1a} descriptor with the respective item and the set of items against which the item needs to be unified. It limits the range of the item set to prevent doing redundant work. Conversely, because the adding of items is serialized, it is guaranteed that no pair of items will be skipped for matching.

[3]Henceforth, we use the term item set to refer to an order imposing iterator.

5 Conclusions and Future research

We have presented a parallelization technique intended for fine-grained parallelism. It minimizes overhead, while eliminating the need for application-dependent fine-grained locking schemes, also making it easier to apply. Only a minimal amount of changes needed to be made to the serialized parts of the parser. Due to the nature of the parsing operations, the distributed parts needed minimal change as well. Finally, all complexities of the scheduler were implemented in a separate, reusable module.

Although originally designed to parallelize a unification-based parser, the technique can easily be extended for probabilistic parsers and other approaches to parsing. In general, the technique is useful for all problems involving highly stochastic task graphs (where both forking factor and task sizes vary wildly). It should be ensured, though, that the total amount of serialized work remains limited. In order to limit the amount of overhead, the number of serialized blocks should be limited as well.

We are currently researching the possibility of exploiting parallelism specific to robust parsing. Robust parsing can increase the computational demands considerably. We also intend to investigate how the grain of parallelism required to effectively exploit parallelism depends on the grammar. The tools we have developed to analyze the Deltra parser can straightforwardly be applied to other parsers.

References

[1] Cynthia Dwork, Paris Kanellakis, and John Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1(1):35–50, 1984.

[2] Hiroto Yasuura. On parallel computational complexity of unification. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'84)*, pages 235–243, Amsterdam, 1984. Institute for New Generation Computer Technology [ICOT].

[3] T. Fujioka, H. Tomabechi, O. Furuse, and H. Iida. Parallelization technique for quasi-destructive graph unification algorithm. In *Information Processing Society of Japan SIG Notes 90-NL-80*, 1990.

[4] H. Tomabechi. Quasi-destructive graph unifications. In *Proceedings of the 29th Annual Meeting of the ACL*, Berkeley, CA, 1991.

[5] Henry S. Thompson. Chart parsing for loosely coupled parallel systems. In Tomita [21], chapter 15.

[6] Henry S. Thompson. Parallel parsers for context-free grammars–two actual implementations compared. In Adriaens and Hahn [13].

[7] Günther Görz, Marcus Kesseler, Jörg Spilker, and Hans Weber. 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 1996.

[8] J.P.M. de Vreught. A practical comparison between parallel tabular recognizers. In *Parsing natural language. Proceedings of the sixth Twente Workshop on Language Technology (TWLT6)*, pages 63–70, 1993.

[9] Marcel P. van Lohuizen. Simulating communication of parallel unification-based parsers. Parallel and Distributed Systems Reports Series PDS-1998-008, Delft University of Technology, 1998.

[10] H.J. Honig. A new double dotted parsing algorithm. Technical Report 94-108, Delft University of Technology, 1994.

[11] R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17(2):416–429, 1969.

[12] Marcel P. van Lohuizen. Minimum-overhead scheduling for fine-grained parallelism. Parallel and Distributed Systems Reports Series To appear soon., Delft University of Technology, 1999.

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

[14] M. Tomita. *Efficient parsing for natural language*. Kluwer Academic Publishers, Boston, MA, 1985.

[15] A. Nijholt. Overview of parallel parsing strategies. In Tomita [21], chapter 14.

[16] Hiroaki Kitano. *Speech-To-Speech Translation: A Massively Parallel Memory-Based Approach*. Natural Language Processing and Machine Translation. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1994.

[17] Peter Neuhaus and Udo Hahn. Restricted parallelism in object-oriented lexical parsing. In *Proc. of the 16th Int. Conf. on Computational Linguistics*, Copenhagen, DK, 5–9 Aug 1996.

[18] H. Kitano. Unification algorithms for massively parallel computers. In *Proceedings of the Second International Workshop on Parsing Technologies*, Cancun, Mexico, 1991.

[19] Enrico Pontelli, Gopal Gupta, Janyce Wiebe, and David Farwell. Natural language multiprocessing: A case study. In *Proceedings of the 15th National Conference on Artifical Intelligence (AAAI '98)*, July 1998.

[20] A.G. Manousopoulou, G. Manis, P. Tsanakas, and G. Papakonstantinou. 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, 1997.

[21] M. Tomita, editor. *Current Issues in Parsing Technology*. Kluwer Academic Publishers, Norwell, MA, 1991.