A Generic Approach to Parallel Chart Parsing with an Application to LinGO

Marcel van Lohuizen

Faculty of Information Technology and Systems Delft University of Technology
Delft, The Netherlands
`mpvl@acm.org`

Abstract

Multi-processor systems are becoming more commonplace and affordable. Based on analyses of actual parsings, we argue that to exploit the capabilities of such machines, unification-based grammar parsers should distribute work at the level of individual unification operations. We present a generic approach to parallel chart parsing that meets this requirement, and show that an implementation of this technique for LinGO achieves considerable speedups.

1 Introduction

The increasing demand for accuracy and robustness for today's unification-based grammar parsers brings on an increasing demand for computing power. In addition, as these systems are increasingly used in applications that require direct user interaction, e.g. web-based applications, responsiveness is of major concern. In the mean time, small-scale desktop multiprocessor systems (e.g. dual or even quad Pentium machines) are becoming more commonplace and affordable. In this paper we will show that exploiting the capabilities of these machines can speed up parsers considerably, and can be of major importance in achieving the required performance.

There are certain requirements the design of a parallel parser should meet. Over the past years, many improvements to existing parsing techniques have boosted the performance of parsers by many factors (Oepen and Callmeier, 2000). If a design of a parallel parser is tied too much to a particular approach to parsing, it may be hard to incorporate such improvements as they become available. For this reason, a solution to parallel parsing should be as general as possible. One obvious way to ensure 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 is basically the approach we will take.

The parser that we will present in this paper uses the LinGO grammar. LinGO is an HPSG-based grammar which was developed at Stanford (Copestake, 2000). It is currently used by many research institutions. This allows our results to be compared with that of other research groups.

In Section 2, we explore the possibilities for parallelism in natural language parsing by analyzing the computational structure of parsings. Section 3 and 4 discuss respectively the design and the performance of our system. Finally, we compare our work with other research on parallel parsing.

2 Analysis of Parsings

To analyze the possibilities for parallelism in computations they are often represented as task graphs. A task graph is a directed acyclic graph, where the nodes represent some unit of computation, called a task, and the arcs represent the execution dependencies between the tasks. Task graphs can be used to analyze the critical path, which is the minimal time required to complete a computation, given an infinite amount of processors. From Brent (1974) and Graham (1969) we know that there exist P-processor schedulings where the execution time T_P is bound as follows:

$$T_P \leq T_1/P + T_\infty, \qquad (1)$$

where T_1 is the total work, or the execution time for the one processor case, and T_∞ is the critical path. Furthermore, to effectively use P processors, the average parallelism $\bar{P} = T_1/T_\infty$ should be larger than P.

The first step of the analysis is to find an appropriate graph representation for parsing computations. According to Caroll (1994), performing a complexity analysis solely at the level of grammars and parsing schemata can give a distorted image of the parsing process in practice. For this reason, we based our analysis on actual parsings. The experiments were based on the fuse test suite, which is a balanced extract from four appointment scheduling (spoken) dialogue corpora (incl. VerbMobil). Fuse contains over 2000 sentences with an average length of 11.6.

We define a task graph for a single parsing computation as follows. First, we distinguish two types of tasks: unification tasks and match tasks. A unification task executes a single unification operation. A match task is responsible for all the actions that are taken when a unification succeeds: matching the resulting edge with other edges in the chart and putting resulting unification tasks on the agenda. The match task is also responsible for applying filtering techniques like the quick check (Malouf et al., 2000). The tasks are connected by directed arcs that indicate the execution dependencies.

We define the cost of each unification task as the number of nodes visited during the unification and successive copying operation. Unification operations are typically responsible for over 90% of the total work. In addition, the cost of the match tasks are spread out over succeeding unification tasks. We therefore simply neglect the cost for match operations, and assume that this does not have a significant impact on our measurements. The length of a path in the graph can now be defined as the sum of the costs of all nodes on

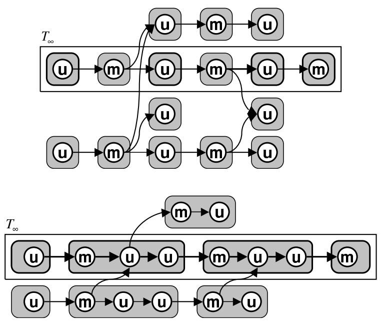

Figure 1: Task graphs for two different approaches to parallel chart parsing.

	T_1	T_∞	$\bar{P}$
Type 1	1014247	3487	187
Average type 2	1014247	11004	54
Worst case type 2	1014247	69300	13

Table 1: Critical path analysis for type 1 and type 2 task graphs (average and worst case).

the path. The critical path length T_∞ can be defined as the longest path between any two nodes in the graph.

The presented model resembles a very fine-grained scheme for distributing work, where each single unification tasks to be scheduled independently. In a straightforward implementation of such a scheme, the scheduling overhead can become significant. Limiting the scheduling overhead is crucial in obtaining considerable speedup. It might therefore be tempting to group related tasks into a single unit of execution to mitigate this overhead. For this reason we also analyzed a task graph representation where only match tasks spawn a new unit of execution. The top graph in Figure 1 shows an example of a task graph for the first approach. The bottom graph of Figure 1 shows the corresponding task graph for the second approach. Note that because a unification task may depend on more than one match task, a choice has to be made in which unit of execution the unification task is put.

Table 1 shows the results of the critical path analysis of both approaches. For the first approach, the critical path is uniquely defined. For the second approach we show both the worst case, considering all possible schedulings, and an average case. The results for T_1, T_∞, and $\bar{P}$ are averaged over all sentences.[1]

The results show that, using the first approach, there is a considerable amount of parallelism in the parsing computations. The results also show that a small change in the design of a parallel parser can have a significant impact on the value for $\bar{P}$. To obtain a speedup of P, in practice, there should be a safety margin between P and $\bar{P}$. This suggests that the first approach is a considerably saver choice, especially when one is considering using more than a dozen of processors.

3 Design and Implementation

Based on the discussion in the preceding sections, we can derive two requirements for the design of a parallel parser: it should be close in design to a sequential parser and it should allow each single unification operation to be scheduled dynamically. The parallel parser we will present in this section meets both requirements.

Let us first focus on how to meet the first requirement. Basically, we let each processor, run a regular sequential parser augmented with a mechanism to combine the results of the different parsers. Each sequential parser component is contained in a different thread. By using threads, we allow each parser to share the same memory space. Initially, each thread is assigned a different set of work, for example, resembling a different part of the input string. A thread will process the unification tasks on the agenda and, on success, will perform the resulting match task to match the new edge with the edges on its chart. After completing the work on its agenda, a thread will 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.

[1]Note that since $\sum T_1 / \sum T_\infty \neq \sum T_1/T_\infty$, the results for $\bar{P}$ turn out slightly lower than might have been expected from the values of T_1 and T_∞.

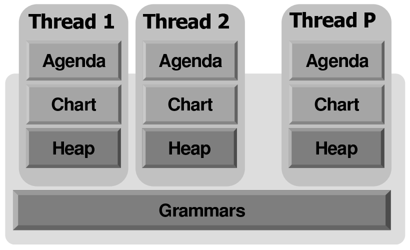

Figure 2: Architecture of MACAMBA.

3.1 Data Structures

Figure 2 shows an outline of our approach in terms of data structures. Each thread contains an agenda, which can be seen as a queue of unification tasks, a chart, which stores the derived edges, and a heap, which is used to store the typed-feature structures that are referenced by the edges. Each thread has full access to its own agenda, chart, and heap, and has read-only access to the respective structures of all other threads. Grammars are read-only and can be read by all threads.

In the communication phase, threads need read-only access to the edges derived by other threads. This is especially problematic for the feature structures. Many unification algorithms need write access to scratch fields in the graph structures. Such algorithms are therefore not thread-safe.[2] For this reason we use the thread-safe unification algorithm presented by Van Lohuizen (2000), which is comparable in performance to Tomabechi's algorithm (Tomabechi, 1991).

Note that each thread also has its own agenda. Some parsing systems require strict control over the order of evaluation of tasks. The distributed agendas that we use in our approach may make it hard to implement such a strict control. 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. Techniques to reduce the synchronization overhead

[2]In this context, thread safe means that the same data structure can be involved in more than one operation, of more than one thread, simultaneously.

```
global shared NrThreadsIdle, Generation, IdleGen

SCHED()
  var threadGen, newWork, isIdle
  threadGen ← Generation ← Generation + 1
  while NrThreadsIdle ≠ $P$ do
      1. newWork ← not ISEMPTY(agenda).
      2. Process the agenda as in the sequential
         case. In addition, stamp each newly de-
         rived $I$ edge by setting $I$.generation to the
         current value for threadGen and add $I$ to this
         thread's edge list.
      3. Examine all the 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 cor-
         responding task to the agenda if it passes
         the filter. If any edge was processed, set
         newWork to true.
      4. if not newWork then
             newWork ← STEAL()
      5. lock GlobalLock
      6. if newWork then
             Generation ← Generation + 1
             threadGen ← Generation
             NrThreadsIdle ← 0
      7. else
             if Generation ≠ IdleGen then
                 isIdle ← false
                 Generation ← Generation + 1
                 threadGen ← IdleGen ← Generation
             elseif threadGen ≠ IdleGen then
                 isIdle ← false
                 threadGen ← IdleGen
             elseif not isIdle then
                 isIdle ← true
                 NrThreadsIdle ← NrThreadsIdle + 1
      8. unlock GlobalLock
```

Figure 3: Scheduling algorithm.

in such a setup can be found in (Markatos and LeBlanc, 1992).

3.2 Scheduling Algorithm

At startup, each thread calls the scheduling algorithm shown in Figure 3. 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 2. After this step, the agenda will be empty. The communication between threads takes place in step 3. Each time a thread executes this step, it will proceed over all the newly derived edges of other threads (foreign edges) and match them with the edges on its own chart (local edges). Checking the newly derived edges of other threads can simply be done by proceeding over a linked list of derived edges maintained by the respective threads. Threads record the last visited edge of the list of each other thread. This ensures that each newly derived item needs to be visited only once by each thread.

As a result of step 3, the agenda may become non-empty. In this case, newWork will be set and step 2 is executed again. This cycle continues until all work is completed.

The remaining steps serve several purposes: load balancing, preventing double work, and detecting termination. We will explain each of these aspects in the following sections. Note that step 6 and 7 are protected by a lock. This ensures that no two threads can execute this code simultaneously. This is necessary because Step 6 and 7 write to variables that are shared amongst threads. The overhead incurred by this synchronization is minimal, as a thread typically iterates over this part only a small number of times. This is because the depth of the derivation graph of any edge is limited (average 14, maximum 37 for the fuse test set).

3.3 Work Stealing

In the design as presented so far, each thread exclusively executes the unification tasks on its agenda. Obviously, this violates the requirement that each unification task should be scheduled dynamically.

In (Blumofe and Leiserson, 1993), it is shown that for any multi-threaded computation with work T_1 and task graph depth T_∞, and for any number P of processors, a scheduling will achieve $T_P \leq T_1/P + T_\infty$ if for the scheduling holds that whenever there are more than P tasks ready, all P threads are executing work. In other words, as long as there is work on any queue, no thread should be idle.

An effective technique to ensure the above requirement is met is work stealing (Frigo et al., 1998). With this technique, a thread will first attempt to steal work from the queue of another thread before denouncing itself to be idle. If it succeeds, it will resume normal execution as if the stolen tasks were its own. Work stealing incurs less synchronization overhead than, for example, a centralized work queue.

In our implementation, a thread becomes a thief by calling STEAL, at step 4 of SCHED. STEAL allows stealing from two types of queues: the agendas, which contain outstanding unification tasks, and the unchecked foreign edges, which resemble outstanding match tasks between threads.

A thief first picks a random victim to steal from. It first attempts to steal the victim's match tasks. If it succeeds, it will perform the matches and put any resulting unification tasks on its own agenda. If it cannot gain exclusive access to the lists of unchecked foreign edges, or if there were no matches to be performed, it will attempt to steal work from the victim's agenda. 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 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. We use an asymmetric mutual exclusion scheme, as presented in (Frigo et al., 1998), to minimize the cost of locking for normal processing and move more of the overhead to the side of the thief.

3.4 Preventing Duplicate Matches

When two matching edges are stored on the charts of two different threads, it should be prevented that both threads will perform 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 2). In addition, 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 3). Obviously, because the value of threadGen is unique for the thread (see step 6), 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 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.

3.5 Termination

A thread may terminate when all work is completed, that is, if and only if the following conditions hold simultaneously: all agendas of all threads are empty, all possible matches between edges have been processed, and all threads are idle. Step 7 of SCHED enforces that these conditions hold before any thread leaves SCHED. Basically, each thread determines for itself whether its queues are empty and raises the global counter NrThreadsIdle accordingly. When all threads are idle simultaneously, the parser is finished.

A thread's agenda is guaranteed to be empty whenever newWork is false at step 7. The same does not hold for the unchecked foreign edges. Whenever a thread derives a new edge, all other edges need to perform the corresponding matches. The following mechanism enforces this. The first thread to become idle raises the global generation and records it in IdleGen. Subsequent idle threads will adopt this as their idle generation. Whenever a thread derives a new edge, it will raise Generation and reset NrThreadsIdle (step 6). This invalidates IdleGen which implicitly removes the idle status from all threads. Note that step 7 lets each thread perform an additional iteration before raising NrThreadsIdle. This allows a thread to check for foreign edges that were derived after step 3 and before 7. Once all work is done, detecting termination

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 2: Execution times for the fuse test suite for various number of processors.

requires at most $2P$ synchronization steps.[3]

3.6 Implementation

The implementation of the system consists of two parts: MACAMBA and CaLi. MACAMBA stands for Multi-threading Architecture for Chart And Memoization-Based Applications. The MACAMBA framework provides a set of objects that implement the scheduling technique presented in the previous section. 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 (Malouf et al., 2000), which, together with the rule check, takes care of filtering over 90% of the failing unification tasks before they are put on the agenda. MACAMBA and CaLi were both implemented in Objective-C and currently run on Windows NT, Linux, and Solaris.

4 Performance Results

The performance of the sequential version of CaLi is comparable to that of PET.[4] In addition, for the single-processor parallel version of CaLi the total overhead incurred by scheduling is less than 1%.

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, for a varying number of processors. Table 2 shows the results of these experiments.[5] 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. The fuse test suite contains a large number of small sentences that are hard to parallelize. These results indicate that deploying multiple processors on all input sentences unconditionally still gives a considerable overall speedup.

The second set of experiments were run on a SUN Enterprise10000 with 64 250 MHz UltraSparc II processors. 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 fewer than several milliseconds. Measuring speedup 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 4. 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. One of the reasons for the decline in speedup after 32 processors is the overhead in the scheduling algorithm. Most notably, the total number of top-level iterations of SCHED increases for larger P. The minimum speedups of around 1 are obtained for, often small, sentences that contain too little inherent parallelism to be parallelized effectively.

Figure 4 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 till 16 processors. Note that super linear speedup is achieved with up to 12 processors, repeatedly for the same set of sentences. Super lin-

[3]No locking is required once a thread is idle.

[4]Respectively, 1231s and 1339s on a 500MHz P-III, where both parsers used the same parsing schema.

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

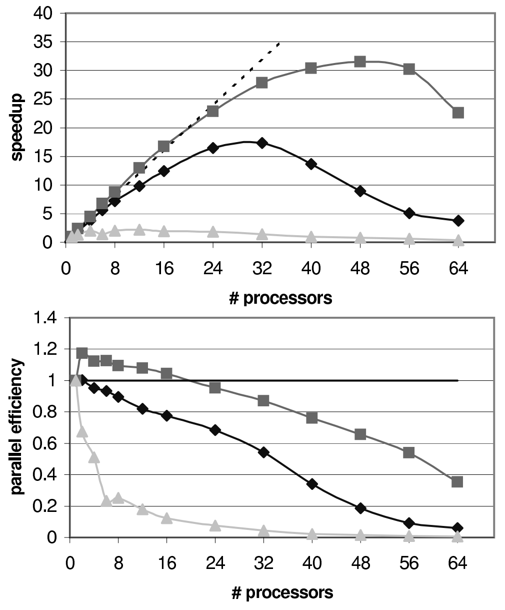

Figure 4: Average, maximum, and minimum speedup and parallel efficiency based on wall clock time.

ear speedup can occur because increasing the number of processors also reduces the amount of data handled by each node. This reduces the chance of cache misses.

5 Related Work

Parallel parsing for NLP has been researched extensively. For example, Thompson (1994) presented some implementations of parallel chart parsers. Nijholt (1994) gives a more theoretical overview of parallel chart parsers. A survey of parallel processing in NLP is given by Adriaens and Hahn (1994).

Nevertheless, many of the presented solutions either did not yield acceptable speedup or were very specific to one application. Recently, several NLP systems have been parallelized successfully. Pontelli et al. (1998) show how two existing NLP applications were successfully parallelized using the parallel Prolog environment ACE. The disadvantage of this approach, though, is that it can only be applied to parsers developed in Prolog.

Manousopoulou et al. (1997) discuss a parallel parser generator based on the Eu-PAGE system. This solution exploits coarse-grained parallelism of the kind that is unusable for many parsing applications, including our own (see also Görz et. al. (1996)).

Nurkkala et al. (1994) presented a parallel parser for the UPenn TAG grammar, implemented on the nCUBE. Although their best results were obtained with random grammars, speedups for the English grammar were also considerable.

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. The speedup ranged up to 8.8 for 20 processors, Parallelism is mainly thwarted by a lack of parallelism in the first phase.

Finally, Ninomiya et al. (2001) developed an agent-based parallel parser that achieves speedups of up to 13.2. It is implemented in ABCL/f and LiLFeS. They also provide a generic solution that could be applied to many parsers. The main difference with our system is the distribution of work. This system uses a tabular chart like distribution of matches and a randomized distribution of unification tasks. Experiments we conducted show that the choice of distribution scheme can have a significant influence on the cache utilization.

It should be mentioned, though, that it is in general hard to compare the performance of systems when different grammars are used.

On the scheduling side, our approach shows close resemblance to the Cilk-5 system (Frigo et al., 1998). It implements work stealing using similar techniques. An important difference, though, is that our scheduler was designed for chart parsers and tabular algorithms in general. These types of applications fall outside the class of applications that Cilk is capable of handling efficiently.

6 Conclusions

We showed that there is sufficient parallelism in parsing computations and presented a parallel chart parser for LinGO that can effectively exploit this parallelism by achieving 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.

Acknowledgements

Thanks to Makino Takaki and Takashi Ninomiya of the Department of Information Science, University of Tokyo, for running the 1–64 processor experiments at their department's computer.

References

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

[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 submitted to SIAM Journal on Computing.

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

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

[Caroll1994] John Caroll. 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.

[Copestake2000] Ann Copestake, 2000. *The (new) LKB system, version 5.2.* from Stanford site.

[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.

[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.

[Malouf et al.2000] Robert Malouf, John Carroll, and Ann Copestake. 2000. Efficient feature structure operations witout 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 LeBlanc1992] E. P. Markatos and T. J. LeBlanc. 1992. 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.

[Nijholt1994] Anton Nijholt. 1994. Parallel approaches to context-free language parsing. In Adriaens and Hahn (1994).

[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.

[Nurkkala and Kumar1994] Tom Nurkkala and Vipin Kumar. 1994. 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.

[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.

[Thompson1994] Henry S. Thompson. 1994. Parallel parsers for context-free grammars–two actual implementations compared. In Adriaens and Hahn (1994).

[Tomabechi1991] H. Tomabechi. 1991. Quasi-destructive graph unifications. In *Proceedings of the 29th Annual Meeting of the ACL*, Berkeley, CA.

[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.

[Yoshida et al.1999] Minoru Yoshida, Takashi Ninomiya, Kentaro Torisawa, Takaki Makino, and Jun'ichi Tsujii. 1999. Proceedings of efficient FB-LTAG parser and its parallelization. In *Proceedings of Pacific Association for Computational Linguistics '99*, pages 90–103, Waterloo, Canada, August.