A Parallel Bottom-up Tomita Parser

Klaas Sikkel, Marc Lankhorst[1]

Dept. of Computer Science, University of Twente,
PO Box 217, 7500 AE Enschede, The Netherlands
(sikkel@cs.utwente.nl)

Abstract

A novel parallel parser is presented, based on a well-known parallelization of Earley's algorithm and an adaptation of Tomita's generalized LR parser. For each word a process parses all constituents starting with that word. Constituents parsed by other processes can be regarded as atomic symbols. Superficially, a process resembles a Tomita parser for a suffix of the sentence. The technicalities are somewhat different, however, and lead to some theoretically interesting improvements.

Practical comparison with a conventional Tomita parser shows a decrease in parsing complexity and an increase in constant factors. That is, the extra costs in communication overhead are offset by the gain in processing power if the sentence is sufficiently large.

1 Introduction

Tomita's generalized LR parser [Tomita85] is a popular parsing algoritm for natural language applications. It combines the ability to handle most context-free grammars with the efficiency of the LR parser. In order to cope with nondeterminism, a set of LR parse stacks is maintained. The different stacks are merged into a graph structure for efficiency.

A couple of parallel Tomita parsers, implemented in a parallel logic programming language, have been presented by Tanaka and Numazaki. Maintaining a graph structured stack would require too much synchronization, therefore they work in parallel on separate copies of linear stacks [Tanaka89] or with tree structured stacks [Numazaki90]. We look at the problem of parallel generalized LR parsing from quite a different angle — taking, in fact, a perpendicular view. Rather than working through the sentence in LR fashion, we remove the left-to-right restriction and introduce processes that parse the sentence purely bottom-up, starting at every word in parallel. Each process runs an adapted Tomita parser, yielding the constituents that start with its own word. Symbols parsed by other processes can be used as atomic entities.

Earley's algorithm [Earley70], [GHR80] scans a sentence from left to right, while keeping track of (partially) recognised constituents in an upper triangular matrix. Like Tomita's algorithm, it is a bottom-up parser with top-down filtering. A straightforward parallelization is obtained by removing the left-to-right restriction, and thereby the top-down filtering. A purely bottom-up algorithm results, in which each column (or each row) of the matrix can be computed in parallel. See, e.g., [Chiang84], [Nijholt91]. Despite the difference in appearance, the algorithms of Earley and Tomita are structurally very similar [Sikkel90]. The algorithm to be presented is the Tomita equivalent of the parallel bottom-up Earley parser in which a processor is allocated to a row of the matrix, hence it is called a Parallel Bottom-up Tomita (PBT) parser.

Parallel processing may save time; it also may cost time due to increased overhead and communication. Thompson presented a parallel chart parser where adding more processors led to an *increase* in computation time [Thompson89]. We tested our parser against Tomita's algorithm, using the test sets given in [Tomita85]. It turns out that PBT is faster for long sentences; for short sentences the increase in processing power does not offset the additional overhead. Furthermore, we found that adding more processors to the PBT

[1]The current adress of the second author is: Dept. of Mathematics and Computer Science, University of Groningen, PO Box 800, 9700 AV Groningen, The Netherlands (lankhors@cs.rug.nl).

parser (up to the number of words in a sentence) always leads to a decrease in computation time.

An interesting theoretical aspect of our PBT parser is that some problems of the standard Tomita parser are eliminated. All context-free grammars can be handled, and nodes in the parse forest for the same constituent are guaranteed to be shared. Furthermore, the parsing tables are easier to construct and much smaller.

The PBT recognizer is described in section 2 and extended to a parser in section 3. In section 4 we discuss empirical comparison between PBT and Tomita's algorithm. Conclusions are summarized in section 5.

2 The PBT recognizer

We define a Parellel Bottom-up Tomita recognizer first, and extend it to a parser in the next section.

Let $a_1 \ldots a_n$ be a sentence according to some context-free grammar G. For technical reasons, a special end-of-sentence marker \$ is added as the $n+1$-th symbol. The recognizer consists of $n+1$ processes $P_0, \ldots, P_n$, communicating asynchronously in a pipeline structure. See Figure 1. It is not necessary, however, that every process runs on a different processor; If there are more words than processors, a single processor can run multiple processes.

The task of process P_i is to recognize all constituents starting with word a_{i+1}, i.e., all $X \in V$ such that $X \Rightarrow^* a_{i+1} \ldots a_j$ for some j. Recognized symbols are tagged with place markers so as to indicate which part of the sentence they span. Thus the sentence will be recognized iff P_0 can recognize a symbol $\langle 0, S, n \rangle$.

Figure 1: A pipeline of processes

Each process P_i starts by recognizing its "own" terminal $\langle i, a_{i+1}, i+1 \rangle$. Symbols that have been recognized by some other process upstream are read from the right neighbour. These are passed on to the left neighbour, while newly recognized symbols are inserted into the stream. In a more sophisticated version, a symbol can be discarded if it can be decided locally that such a symbol is irrelevant for the remainder of the pipeline.

Each P_i uses two data structures: a precomputed parsing table and a graph structured stack in which (partially) recognized constituents are stored. It is called "graph structured stack" as in Tomita's algorithm. It is not really a stack, though, as nothing gets ever deleted. The stack contains *state vertices* labelled with parser states and *symbol vertices* labelled with recognized symbols.

Rather than giving a formal definition, we will explain the algorithm by working through an example. The grammar G is defined by

(1) $S \rightarrow NP\ VP$
(2) $NP \rightarrow \text{det}\ {}^*n$
(3) $NP \rightarrow {}^*n$
(4) $NP \rightarrow NP\ PP$
(5) $PP \rightarrow {}^*p\ NP$
(6) $VP \rightarrow {}^*v\ NP$
(7) $VP \rightarrow VP\ PP$

The parsing table is shown in Figure 2, its construction will be discussed later.

	action	*goto*								
		$^*d$	$^*n$	$^*p$	$^*v$	S	NP	PP	VP	\$
0		4	5	6	7	1	2		3	
1										*acc*
2								9	8	
3								10		
4			11							
5	$r3$									
6							12			
7							13			
8	$r1$									
9	$r4$									
10	$r7$									
11	$r2$									
12	$r5$									
13	$r6$									

Figure 2: The PBT parsing table for G

The *action* column tells in which state a reduction can be carried out, and which production is being reduced. Shift actions are not explicitly mentioned, a symbol can be shifted in a particular state if a successor state is shown in the *goto* table. Acceptation is disguised as a shift; the sentence is accepted iff \$ is shifted.

As an example we take the canonical sentence *I saw the man with a telescope*. We single out P_1 and follow the construction of its stack. It's task

is to recognize all constituents startin with *saw*. The stream of symbols that is read from P_2 in due course is

$$\langle 2, NP, 4\rangle,\ \langle 4, PP, 7\rangle,\ \langle 2, NP, 7\rangle,\ \langle 7, \$, 8\rangle.$$

We start with an empty stack, represented by a single state vertex labelled 0. First, P_1's terminal symbol $\langle 1, *v, 2\rangle$ is shifted. That is, a symbol vertex with that label is created, followed by a state vertex labelled 7 (the new state according to the goto table). No reduction can be made, so we read $\langle 2, NP, 4\rangle$ from the pipe. In state 7 this can be shifted. The new state is 13, requiring action *re*6. Using rule (6) we rewrite $\langle 1, *v, 2\rangle\langle 2, NP, 4\rangle$ into $\langle 1, VP, 4\rangle$. We do *not* delete the reduced branch from the stack, as it might still be needed. We simply start a new branch from the initial node, shifting the completed *VP* just as if it had been read from the pipe, See Figure 3.

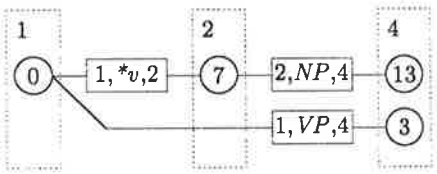

Figure 3: The stack after reducing $\langle 1, VP, 4\rangle$

It is important to notice that state nodes are grouped into sets, which are identified by positions in the sentence. We may jump back and forth between positions, making extensions wherever appropriate.

The next symbol, $\langle 4, PP, 7\rangle$, is shifted in state 3 (at position 4) and $\langle 1, VP, 4\rangle\langle 4, PP, 7\rangle$ is reduced to $\langle 1, VP, 7\rangle$.

Note that $\langle 4, PP, 7\rangle$ could not be shifted from state 13 — there is no entry in the goto table — although $\langle 2, NP, 4\rangle\langle 4, PP, 7\rangle$ is reducible to a compound *NP*. This is because P_1 only creates new symbols that start at position 1. As we read the next symbol, it turns out that $\langle 2, NP, 7\rangle$ has been created by P_2 already. It is shifted at position 2. Subsequently we can reduce a verb phrase $\langle 1, VP, 7\rangle$. This symbol is already present in the stack and need not be added again.

The last symbol, $\langle 7, \$, 8\rangle$, cannot be shifted anywhere. It also signals the end of the stream, hence P_1 has finished its task. The final parse stack is shown in Figure 4.

Symbols are sent on to the left neighbour as

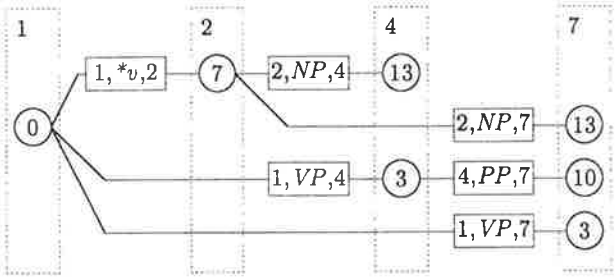

Figure 4: The final stack of P_1

soon as they are read or created, in order to minimize waiting time. Some ordering requirements must be made, however, so as to guarantee a proper functioning of the algorithm. In particular, a symbol $\langle j, Y, k\rangle$ must have been preceded by all symbols $\langle i, X, j\rangle$ for $i \leq j \leq k$, otherwise the state vertex on which $\langle j, Y, k\rangle$ is to be shifted might not yet be present. This requires some careful handling when multiple nullable symbols $\langle j, X, j\rangle$ and $\langle j, Y, j\rangle$ are present. In all other cases, the ordering requirements are satisfied naturally.

If all symbols created by all processes are passed down the pipeline, this may result in a communication bottleneck. With some additional effort most symbols can be discarded that are not needed further down the pipeline. For example: P_2, with terminal $\langle 2, \mathrm{det}, 3\rangle$, will receive the noun phrase $\langle 3, NP, 7\rangle$, *man with a telescope*. But $\langle 2, \mathrm{det}, 3\rangle\langle 3, NP, 7\rangle$ cannot be part of a sentential form, hence the latter symbol can be discarded. Such knowledge can be compiled into "communication tables", indicating in which cases symbols can be safely discarded. For more detailed treatment of filtering see [Lankhorst91].

Construction of a PBT parsing table resembles the construction of a conventional LR(0) parsing table [Aho77]. States, represented by integers, in fact consist of *sets of LR(0) items*. The initial state 0 is defined as

$$\begin{array}{ll} \{\ S' \rightarrow \bullet S\,\$, & NP \rightarrow \bullet NP\ PP, \\ \ \ S \rightarrow \bullet NP\ VP, & PP \rightarrow \bullet *p\ PP, \\ \ \ NP \rightarrow \bullet \mathrm{det}\ *n, & VP \rightarrow \bullet *v\ NP, \\ \ \ NP \rightarrow \bullet *n, & VP \rightarrow \bullet VP\ PP\ \} \end{array}$$

That is, initially we are ready to recognize any constituent. The dot indicates how far we have proceeded in scanning each of of the right-hand sides. For each symbol after a dot, an entry in

the goto table and a new state must be defined. If a constituent starts with a NP, for example, we move to state 4, being

$$\{\ S \rightarrow NP.VP,\quad NP \rightarrow NP.PP\ \}$$

In a conventional LR table, one should add the PREDICT sets of VP and PP to this state; not so in a PBT table! if a PP is to follow the NP, a PP symbol will arrive in due course, created by another processor further upstream. Such a PP symbol leads to a state $\{NP \rightarrow NP\ PP.\}$. This is state 9 in the table, in which the reduction into a compound NP is called for.

The small simplification by leaving out the PREDICT set makes a dramatic difference for the size of the table. As table size we take the number of *non-empty* table entries; Goto tables are usually large and sparse, therefore they are represented as an array of lists rather than a matrix. For large grammars (as III and IV in [Tomita85]) the PBT tables are typically an order of magnitude smaller than Tomita's tables. Table size and computation time is linear in the size of the grammar. Hence, in situations where the size of a standard LR table is prohibitive or where the grammar is often changed, a sequentialized version of PBT could be run on a single workstation.

3 The PBT parser

The PBT recognizer can be easily extended into a parser. As with Tomita's algorithm, the parser yields a *packed shared forest*, a graph structure in which common sub-parses are shared. The format in which this forest is delivered is a *parse list*, containing an entry for each node, with a list of pointers to child nodes (if any). Ambiguities are represented by multiple lists of child nodes (called *sub-nodes* by Tomita). If the sentence can be parsed, a pointer to the root node is given.

In order to compute the parse list in a distributed fashion, one technical adjustment need be made: the left place marker of a symbol is annotated with its label in the parse list. Whenever a processor reduces a symbol, a node is added to its part of the parse list. If the same symbol is reduced a second time, a new sub-node is added to the already existing node. The completed parse list for the example sentence is shown in Figure 5, the root node is $\langle 0.4, S, 7\rangle$.

The parse forest is not identical to the one produced by Tomita's algorithm. For every triple

symbol	*children*	
$\langle 6.1, {}^*n, 7\rangle$		
$\langle 6.2, NP, 7\rangle$	(6.1)	
$\langle 5.1, \text{det}, 6\rangle$		
$\langle 5.2, NP, 7\rangle$	(5.1, 6.1)	
$\langle 4.1, {}^*p, 5\rangle$		
$\langle 4.2, PP, 5\rangle$	(4.1, 5.2)	
$\langle 3.1, {}^*n, 4\rangle$		
$\langle 3.2, NP, 4\rangle$	(3.1)	
$\langle 3.3, NP, 7\rangle$	(3.2, 4.2)	
$\langle 2.1, \text{det}, 3\rangle$		
$\langle 2.2, NP, 4\rangle$	(2.1, 3.1)	
$\langle 2.3, NP, 7\rangle$	(2.2, 4.2)	
$\langle 1.1, {}^*v, 2\rangle$		
$\langle 1.2, VP, 4\rangle$	(1.1, 2.2)	
$\langle 1.3, VP, 7\rangle$	(1.1, 2.3)	(1.2, 4.2)
$\langle 0.1, {}^*n, 1\rangle$		
$\langle 0.2, NP, 1\rangle$	(0.1)	
$\langle 0.3, S, 4\rangle$	(0.2, 1.2)	
$\langle 0.4, S, 7\rangle$	(0.2, 1.3)	

Figure 5: The parse list, root is 0.4

$\langle i, X, j\rangle$ such that $X \Rightarrow^* a_{i+1} \ldots a_j$, a node is added to the forest. Tomita, using the top-down filtering implicit in shift/reduce parsing, only creates a node if the additional condition $S \Rightarrow^* a_1 \ldots a_j \gamma$ is satisfied for some $\gamma \in V^*$. Hence our forest contains more unreachable nodes than Tomita's. On the other hand, if X does produce $a_{i+1} \ldots a_j$, this is represented in our forest with a *unique* node, possibly containing multiple sub-nodes. Thus the structure of our parse forest is fully specified. For Tomita's algorithm is it very hard to precisely specify the forest that will be delivered; a symbol spanning some specific part of the sentence is *usually* represented by a single node. Sharing fails, however, if identical symbol vertices on the stack are followed by different state vertices.

A more substantial improvement upon Tomita's algorithm is the acceptance of arbitrary context-free grammars. Apart from cyclic grammars, there is a class of non-cyclic context-free grammars that cannot be handled by Tomita's algorithm. This problem has been identified in [Nozohoor89]. We call a grammar *pseudo-cyclic* if there is a terminal A such that $A \Rightarrow^* \alpha A \beta$, with $\alpha \Rightarrow^+ \varepsilon$ and $\beta \not\Rightarrow^* \varepsilon$. Consider the pseudo-cyclic grammar

$$\{S \rightarrow A\,S\,b,\quad S \rightarrow x,\quad A \rightarrow \varepsilon\}.$$

If the string starts $xb\ldots$, how many A's must be

reduced before the x is shifted? Tomita's algorithm, anticipating an arbitrary number of b's, creates infinitely many A's for a start. Nozohoor-Farshi proposes to create a *loop* in the graph-structured stack; as many A's as needed can be used by unrolling the loop arbitrarily often. In the PBT parser the problem with pseudo-cyclic grammars simply does not occur. A single symbol $\langle 0, A, 0\rangle$ — or, to be precise, the state vertex following it's symbol vertex — can be used to shift any $\langle 0, S, k\rangle$ and subsequent $\langle k, b, k{+}1\rangle$ on.

Cyclic grammars are also parsed in a natural way, without the need for extra sophistication. Consider the grammar $\{S \rightarrow S,\ S \rightarrow a\}$, and the sentence a. When $\langle 0, S, 1\rangle$ is recognized, it is reduced to $\langle 0, S, 1\rangle$, which is already present, and need not be added again. Thus the parser will add the corresponding node as a sub-node to *itself*. The complete parse list is shown in Figure 6.

symbol	*children*
$\langle 0.1, a, 1\rangle$	
$\langle 0.2, S, 1\rangle$	(0.2), (0.1)

Figure 6: The parse for a, $G = \{S \rightarrow S|a\}$

Dealing with arbitrary grammars and optimal node sharing for Tomita's algorithm are discussed in Chapter 1 of [Rekers92]; in our approach both features come about naturally.

4 Empirical results

The PBT algorithm has been tested in a series of experiments in which parallel execution was simulated on a single workstation, In this way we could experiment with an arbitrary number of (simulated) processors.

The simulation set-up is as follows. Each (virtual) process is run consecutively. The stream of symbols is stored internally, rather than written to a pipe. When the next virtual process is started, the clock is reset. For every (simulated) read and write an extra processing time of 1 ms is counted. Each symbol that is sent from one virtual process to another is timestamped. When a process receives a symbol with a timestamp later than its own time, the clock is updated and the waiting time accounted for.

We implemented PBT in the language C and re-implemented Tomita's algorithm so as to ensure compatibility. We have not attempted to optimize run-time efficiency at the expense of straightforwardness. The timing experiments have been conducted on a Commodore Amiga because of its accurate timing capabilities.

The grammars and example sentences are the ones given in [Tomita85]. Grammar I is the toy grammar of our example. Grammars II, III and IV have 42, 223 and 386 rules, respectively. Sentence set A contains 40 sentences, taken from actual publications; set B is constructed as $*n\ *v$ det $*n(*p$ det $*n)^{k-1}$ with k ranging from 1 to 13. In Figures 7 and 8 the timing results for set B and grammars III and IV are plotted on a double logarithmic scale. These figures show that gain in speed due to parallelisation outweighs the additional communication overhead only if a sentence is sufficiently long. An exact break-even point cannot be given, as it depends on the grammar, the sentence, the characteristics of the parallel architecture and the implementation.

Similarly, Figure 8 shows that the extra overhead for filtering pays off only if the sentence is not too small. We could tip the balance somewhat more in favour of PBT by improving the filter. In the program that was used to produce these plots, the filter has a computational complexity linear in the size of the grammar. In retrospect, this could have been handled rather more efficiently. Adding sophistication to handling the graph structured stack and parsing table look-up could improve the performance in absolute terms; relatively it would make less difference, however, as all programs would benefit from it.

Testing sentence set A produces plots of a more varied nature, as sentences of comparable length may differ a lot in complexity. Using linear regression analysis, we found the overall trend to be similar to the results for set B. For reasons of space, we refer to [Lankhorst91] for more details.

The complexity of a parsing algorithm can be measured as a function of the length of the input sentence. For formal languages this makes sense, as strings (i.e., computer programs) can be very long indeed. For natural languages this is a rather doubtful measure. The size of the grammar, usually *much* larger than the average sentence, is constant and therefore considered irrelevant. Nevertheless, sentence set B shows the complexity of the algorithms rather nicely, because of the combinatorial explosion of *PP* attachment

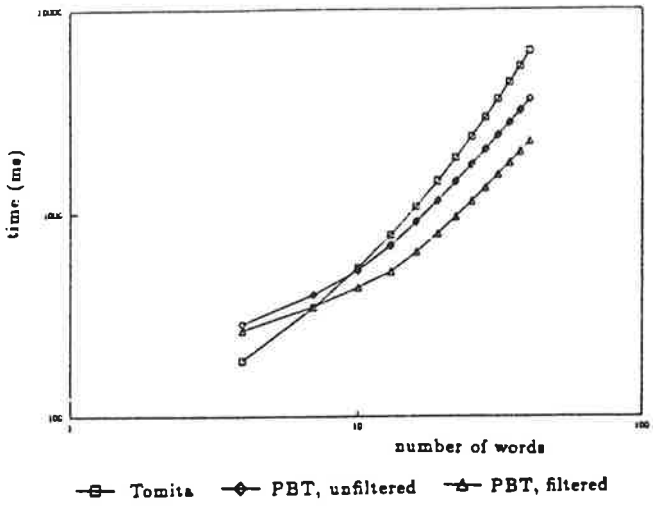

Figure 7: Sentence set B and grammar III

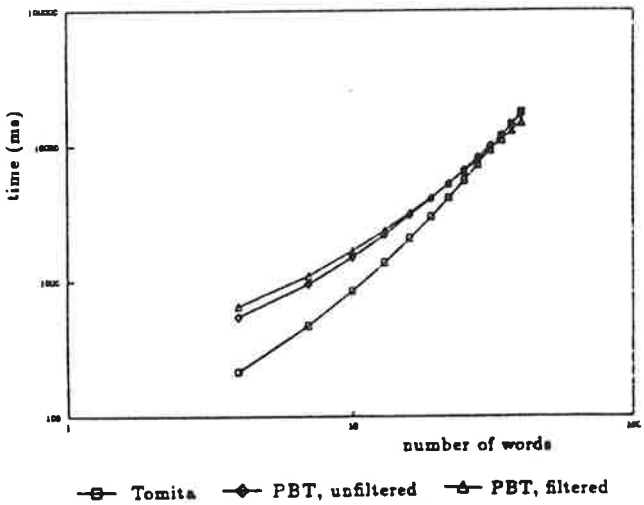

Figure 8: Sentence set B and grammar IV

ambiguities. Moreover, constant factors as discussed above are abstracted from. For set B and grammars III and IV we estimated the asymptotic complexity. These figures, for what they are worth, are shown in Figure 9. Similar computations for sentence set A confirm the trend that the complexity of PBT, using n parallel processes, is roughly $O(\sqrt{n})$ better than Tomita's algorithm.

Finally, we have estimated the speed of the PBT algorithm as a function of the number of processors. The 37 processes for the sentence 13 of set B have been allocated to any number of processors ranging from 1 to 37, with the processes evenly distributed over the processors. Let p be the number of processors, and k such that

algorithm	*grammar*	
	III	IV
Tomita	$O(n^{2.21})$	$O(n^{2.61})$
PBT, unfiltered	$O(n^{1.62})$	$O(n^{2.19})$
PBT, with filtering	$O(n^{1.50})$	$O(n^{1.86})$

Figure 9: Asymptotic complexity for set B

$k \leq 37/p < k+1$. The higher ranked processes are groups in clusters of $k+1$, the lower ranked in clusters of k. The results are shown in figure 10. The decline is sharpest when k is decreased, i.e., the processor handling $P_0, P_1, \ldots$ is relieved of one of its processes.

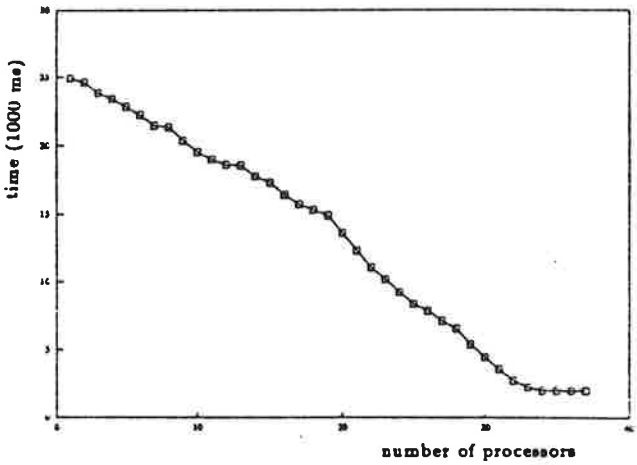

Figure 10: Performance vs. number of processors

5 Conclusions

We have presented a parallel adaptation of a generalized LR parser that works purely bottom-up. A nice theoretical improvement upon Tomita's algorithm is the ability to handle arbitrary context-free grammars without additional effort. Also, a specification of the shared parse forest can be given easily.

The size and computation time of the parsing table is linear in the size of the grammar. This makes a (possibly sequentialized) PBT parser an interesting candidate for a linguist's workbench in which the grammar is often changed.

Experiments based on the test sets provided in [Tomita85] indicate that parallelization pays off for sufficiently long sentences. A conventional

Tomita parser is faster for short sentences. Furthermore, a decrease in number of processors allocated to the PBT parser leads to an increase in computation time, and reversed.

References

[Aho77] A.V. AHO, J.D. ULLMAN, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass. (1977).

[Chiang84] Y.T. CHIANG, K.S. FU, Parallel Parsing Algorithms and VLSI implementations for Syntactic Pattern Recognition, *Transactions on Pattern Analysis and Machine Intelligence*, **PAMI-6** (1984) 302–314.

[Earley70] J. EARLEY, An Efficient Context-Free Parsing Algorithm, *Communications of the ACM* **13** (1970) 94–102.

[GHR80] S.L. GRAHAM, M.A. HARRISON, W.L. RUZZO, An Improved Context-Free Recognizer, *Transactions on Programming Languages and Systems* **2** (1980) 415–462.

[Lankhorst91] M.M. LANKHORST, K. SIKKEL, *PBT: A Parallel Bottom-up Tomita Parser*, Memoranda Informatica 91-69, University of Twente, Enschede, the Netherlands (1991).

[Nijholt91] A. NIJHOLT, The Parallel Approach to Context-Free Language Parsing, in: U. HAHN, G. ADRIAENS (Eds.), *Parallel Models of Natural Language Computation*, Ablex Publishing Co., Norwood, N.J. (1991).

[Nozohoor89] R. NOZOHOOR-FARSHI, Handling of Ill-designed Grammars in Tomita's Parsing Algorithm, *Proc. Int. Workshop on Parsing Technologies*, Carnegie Mellon University, Pittsburgh, Pa. (1989) 182–192.

[Numazaki90] H. NUMAZAKI, H. TANAKA, A New Parallel Algorithm for Generalized LR Parsing, *Proc. 13th Int. Conf. on Computational Linguistics (COLING'90)*, Helsinki (1990) Vol. 2, 304–310.

[Rekers92] J. REKERS, *Parser Generation for Interactive Environments*, Ph.D. Thesis, University of Amsterdam (1992).

[Sikkel90] K. SIKKEL, *Cross-Fertilization of Earley and Tomita*, Memoranda Informatica 90-69, University of Twente, Enschede, the Netherlands (1990).

[Tanaka89] H. TANAKA, H. NUMAZAKI, Parallel Generalized LR Parsing based on Logic Programming, *Proc. Int. Workshop on Parsing Technologies*, Carnegie Mellon University, Pittsburgh, Pa. (1989) 329–338.

[Thompson89] H.S. THOMPSON, Chart Parsing for Loosely Coupled Parallel Systems, *Proc. Int. Workshop on Parsing Technologies*, Carnegie Mellon University, Pittsburgh, Pa. (1989) 320–328.

[Tomita85] M. TOMITA, *Efficient Parsing for Natural Language*, Kluwer Academic Publishers, Boston, Mass. (1985).